João Miguel Ferreira Pecorelli

Licenciatura em Engenharia Eletrotécnica e de Computadores

Simulador de Linguagem em Texto Estruturado para Autómato TSX3721

Dissertação para obtenção do Grau de
Mestre em Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Luís Brito Palma,
Professor Auxiliar, Universidade Nova de Lisboa

Co-orientador: Doutor João Almeida das Rosas,
Professor Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Doutor Paulo de Sousa Gil

Arguente: Doutor João Alves Martins

Dezembro, 2014

Simulador de Linguagem em Texto Estruturado para Autómato TSX3721

AGRADECIMENTOS

A realização desta dissertação de Mestrado contou com importantes apoios e incentivos sem os quais não se teria tornado realidade e pelos quais estarei eternamente grato.

Quero expressar o meu sincero agradecimento a todas as pessoas que de alguma forma contribuíram para a conceção e conclusão do projeto final que culminou nesta dissertação.

Ao Professor Doutor Luís Brito Palma, agradeço todo o apoio, compreensão que me dispensou, a criação de condições para o desenvolvimento deste trabalho, pelas opiniões, críticas e o constante incentivo e entusiasmo ao longo desta jornada.

Ao Professor Doutor João Rosas, pela sua orientação, disponibilidade, total colaboração no solucionar de dúvidas e problemas de implementação ao longo da realização do trabalho, um muito obrigado também.

Aos meus amigos e colegas de curso, em particular, aos colegas Bruno Ferreira, Carlos Posse, Diogo Silva, Diogo Teixeira, Filipe Carrasquinho, Gonçalo Domingues, Gonçalo Mestre, Hugo Santos, Hugo Viana, João Santos, Leonardo Miúdo, Ricardo Bernardo e Sérgio Saro, o meu agradecimento pela sua amizade, companheirismo, profissionalismo e por ajudarem a preencher a minha vida com bons momentos.

Aos meus grandes amigos Bruno Vaz, Diogo Dias, Filipe Santos, João Amaral, Miguel Silva, Nuno Figueiredo, Nuno Pais, Ricardo Antunes, Rui Rodrigues e Vasco Silva, um muito obrigado por todos os grandes e bons momentos que passei nestes anos de amizade.

Um especial agradecimento à minha amiga, companheira, namorada Ana Barbosa, por todo o apoio, paciência, disponibilidade, compreensão e incentivo ao longo de toda esta fase importante da vida.

Ao terminar esta etapa, não poderia deixar de agradecer profundamente aos meus pais, que sem eles nada disto seria possível, e a toda a minha família pelo entusiasmo e motivação ao longo de todo o meu percurso académico.

RESUMO

Os controladores lógicos programáveis (PLC) são muito ,e cada vez mais, utilizados na indústria. Este tipo de equipamento, além de ser inerentemente caro, pode causar situações perigosas e perda de produtividade caso sejam incorretamente programados. Existem programas de alguns fabricantes que permitem a simulação de linguagens utilizadas na programação de autómatos (segundo a norma internacional IEC 61131-3), mas uma simulação da linguagem em Texto Estruturado (ST) em particular, é complicada e de difícil acesso. O principal objetivo deste projeto é a realização de um simulador em tempo real, capaz de realizar testes e simulações prévias de código de Texto Estruturado, a testar posteriormente no autómato TSX Micro 3721 da *Schneider*. É necessário a construção de um compilador e interpretador de linguagens de programação, para realização do simulador e interface gráfica. Os dois sistemas utilizados para gerar as regras formais gramaticais e de produção em linguagens de programação são o Lex e o Yacc. O Lex gera um analisador léxico dividindo o ficheiro de linguagem em texto estruturado em *tokens* (símbolos significativos). O Yacc, através desses símbolos recebidos, forma as regras de produção e a respetiva estrutura hierárquica do programa. A partir da interpretação desta estrutura gerada é possível traduzir ou compilar qualquer linguagem de programação, neste caso o texto estruturado, e criar um simulador correspondente numa outra linguagem. O simulador recebe o código ST e através de funções definidas pelo sistema de compilação, interpretação e tradução, reconhece as instruções em linguagem de texto estruturado correspondente e realiza o conjunto de ações propostas. Analisando os testes no autómato e os resultados apresentados pela interface, podemos concluir que é possível realizar compiladores e respetivos simuladores, utilizando análises gramaticais das linguagens de programação, em particular, podemos concluir que o simulador para linguagem em Texto Estruturado foi realizado com sucesso.

Palavras-chave: Linguagem em Texto Estruturado (ST), Controlador Lógico Programável (PLC), Compiladores, Simulador, Análise Léxica, Análise Sintática.

ABSTRACT

The programmable logic controllers (PLCs) are increasingly widely used in industry. In addition to their equipment are inherently expensive, can also result in loss of productivity and cause dangerous conditions when they are not properly used. There are software programs allow the simulation of programming languages used in the PLC program (IEC 61131-3 norm), but the simulation of Structured Text language (ST), in particular, is complicated and difficult to access. The main objective of this project is the fulfillment a real-time simulator, able to perform preliminary tests and simulations of ST code, in order to test later, in the PLC TSX3721 (Schneider).

The simulator and the respective graphical user interface (GUI) are written in a different programming language (C#) , comparing with the testing ST code. For this reason, it is necessary to build a compiler and an interpreter of programming languages. The two used systems that generate the context-free grammar (CFG) or the gramatical and production rules on programming languages are Lex & Yacc. Lex generates a lexical analyser splitting the source file in tokens (atomic parse element). Yacc with the received tokens generates the production rules and the gramatical rules (CFG), ie the hierarchical structure of the program. From the interpretation of this structure it is possible translate or compile a programming language (ST) and build a corresponding compiler in another programming language.

The simulator receives the ST code and through by defined functions recognises the corresponding ST statements and performs a set of proposed instructions. The defined functions are generated and built in compiler system.

It is possible to conclude that the simulator of the structured text language was realised with success, analysing the tests on PLC and the GUI's presented results. In particular, it is possible built compilers and respective simulators using grammatical analysing by CFG of programming languages.

Keywords: Programmable Logic Controllers (PLC), Structured Text Language (ST), Compilers, Simulator, Lexical Analyser, Syntax Analyser.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGENS

ACRÓNIMOS

BNF	*Backus–Naur Form*
CFG	*Context-Free Grammar*
CPU	*Central Processing Unit*
DLL	*Dynamic-Link Library*
DMZ	*Demilitarized Zone (Perimeter Network)*
FBD	*Function Block Diagram*
GUI	*Graphical User Interface*
IDE	*Integrated Development Environment*
IEC	*International Electrotechnical Commission*
IL	*Instruction List*
IP	*Internet Protocol*
LALR	*Look-Ahead Left-Right*
LD	*Ladder Diagram*
LED	*Light-Emitting Diode*
LEX	*Lexer (Scanner)*
PCMCIA	*Personal Computer Memory Card International Association*
PID	*Proportional-Integral-Derivative*
PLC	*Programmable Logic Controller*
SFC	*Sequential Function Chart*
ST	*Structured Text*
TCP	*Transmission Control Protocol*
VDC	*Virtual Design and Construction*

YACC *Yet Another Compiler Compiler*

CAPÍTULO 1

INTRODUÇÃO

Este capítulo resume os principais temas de interesse na conceção da dissertação para a construção de um simulador de linguagem em texto estruturado para o autómato TSX3721 *Schneider*.

São apresentados os problemas e as motivações que levaram ao interesse em concretizar este projeto, assim como os principais objetivos e contribuições referentes à sua realização.

A organização da tese é também apresentada e estruturada, salientando os pontos mais importantes que constituem o modelo e o plano de realização dos respetivos objetivos inicialmente propostos.

INTRODUÇÃO

1.1 Enquadramento e Motivação

A linguagem ST (*Structured Text*), é uma linguagem textual estruturada de alto nível com recursos semelhantes às linguagens "Pascal"e “C”. Como uma das linguagens utilizadas em ambiente industrial e no mundo da automação é fundamental realizar um trabalho quase perfeito para se obter um desempenho e rendimento dentro dos objetivos inicialmente propostos. Como linguagem de programação de alto nível, o texto estruturado é muitas vezes procurado para programação de autómatos ou controladores lógicos programáveis (PLC’s).

As aplicações básicas de *hardware* típicas dos autómatos, podem apresentar aspetos físicos com bastante diferença entre si e com desempenhos variáveis, no entanto, os elementos que os constituem são fundamentalmente os mesmos, resultando em funcionalidades idênticas, independentemente do fabricante e da série do produto. Os autómatos podem representar sérios riscos, quando não são programados convenientemente.

Um dos problemas detetados na indústria, ou em casos particulares do uso de autómatos, como em universidades, é a falta ou a dificuldade em simular e testar previamente os códigos desenvolvidos, manifestando-se particularmente para a linguagem em texto estruturado. Na maioria dos casos, a única alternativa consiste em realizar o teste diretamente na máquina, aumentado drasticamente o risco de ocorrência de avarias nos sistemas.

1.2 Objetivos e Contribuições

O principal objetivo proposto neste projeto será a garantia de uma simulação prévia do código utilizado para programar o autómato TSX3721 (especificação do autómato em 3.2.1). O laboratório utilizado para testes do projeto não possui nenhum simulador para texto estruturado, sendo os testes realizados diretamente no autómato. Para se poder simular previamente é necessário a realização de um segundo código em paralelo, que numa outra linguagem de programação e com o mesmo raciocínio de funcionamento do kit instalado (especificação do Kit a implementar em 3.2.2) no PLC utilizado (autómato TSX3721) , é criada uma interface gráfica para utilizador (3.3.4). Esta interface permite a

simulação em tempo real do funcionamento do kit utilizado.

Sem um sistema de interpretação, reconhecimento e simulação prévia de linguagem em texto estruturado, sempre que se desejar testar um processo complexo ou um conjunto de vários e diferentes processos, é necessário a criação repetida de linguagens de programação diferentes das programadas nos processos, e simulá-las separadamente até se obter o objetivo desejado. Esta metodologia e mecanismo são demasiado complexos, para além de possuir uma enorme probabilidade de ocorrência de erros durante a sua programação.

A utilização do compilador, constitui uma enorme vantagem em relação à metodologia proposta anteriormente. Todos os códigos tornam-se praticamente diretos e facilmente simuláveis através de um simulador, bastando apenas a criação do código na linguagem do controlador programável, neste caso a linguagem em texto estruturado (ST - *Structured Text*), definir toda a sua estrutura gramatical resultante através da criação de analisadores léxicos e sintáticos, e por fim, com o desenvolvimento de uma interface gráfica e com o significado semântico das regras gramaticais anteriormente especificadas, gerar o simulador.

Resumidamente, as três principais etapas e objetivos do projeto que contribuem para um melhoramento significativo do desempenho, velocidade e custos envolvidos, são:

1. Criação do código em linguagem em texto estruturado (ST);
2. Geração de uma estrutura hierarquizada gramaticalmente da linguagem de texto estruturado;
3. Desenvolvimento semântico a partir da gramática gerada e construção do respetivo simulador por meio de uma interface gráfica para utilizador (GUI).

1.3 Organização da Tese

A dissertação está organizada em 5 capítulos principais:

1. Introdução
2. Estado de arte
3. Arquiteturas propostas, tecnologias e implementação
4. Resultados experimentais
5. Conclusão

No capítulo 1 destinado à introdução são abordados os principais problemas encontrados e as respetivas motivações para o desenvolvimento do projeto, tal como as propostas de solução. São portanto apresentados os objetivos concretos no desenvolvimento do trabalho e as respetivas contribuições inerentes à sua conclusão.

O estado da arte no segundo capítulo representa uma visão geral dos temas abordados na realização do projeto. Tendo em conta a sua complexidade intrínseca, em relação às especificações de alguns dos assuntos estudados neste projeto, são introduzidos apenas as principais características dos temas abordados, o seu funcionamento geral, o seu enquadramento histórico, e a resumida explicação de especificações e caracterizações das tecnologias utilizadas.

O terceiro capítulo é referente a toda a caracterização da arquitetura utilizada, metodologia proposta, especificações técnicas da tecnologia utilizada, e por fim, toda a implementação prática dos elementos constituintes à realização do projeto. De modo a não tornar as explicações da implementação demasiado complexas e de difícil interpretação, as principais especificações teóricas das tecnologias utilizadas, são especificadas com mais detalhe também neste capítulo 3.

O capítulo 4 é destinado à apresentação dos resultados experimentais, aos testes preliminares e validação dos resultados obtidos.

Por fim o capítulo 5, referente às conclusões de toda a realização do projeto, ao sumário de todo o trabalho desenvolvido e à conclusão final inerente aos resultados obtidos face aos propostos inicialmente.

CAPÍTULO 2

ESTADO DA ARTE

Este capítulo destina-se à arquitetura e apresentação teórica dos temas abordados ao longo do projeto, enumerando alguns dos seus conceitos gerais, históricos e vantajosos perante a devida utilização dos mesmos.

Na secção (2.1) de automação industrial (pág. 9), são apresentados os conceitos e conhecimentos fundamentais abordados pelas arquiteturas utilizadas em relação aos autómatos (PLCs), linguagens de programação em autómatos e respetivas características e métodos nos quais os seus intervenientes se aplicam.

Para a construção de Compiladores, na secção 2.2 (pág. 13) são abordados, de uma forma geral e sintetizada, as metodologias mais utilizadas nos processos de construção de compiladores, tradutores de linguagens e análises léxicas, sintáticas e semânticas de diferentes linguagens de programação.

Conforme referido, cada secção é iniciada com um breve enquadramento histórico abordando as principais etapas de evolução e criação dos respetivos temas especificados ao longo do projeto.

ESTADO DE ARTE

2.1 Automação Industrial

2.1.1 Enquadramento Histórico

Os computadores digitais, que são dispositivos programáveis de uso geral, rapidamente foram aplicados para o controlo de processos industriais. Em pouco tempo, os computadores necessitaram de programadores especializados, de controlo ambiental de funcionamento rigoroso de temperatura, limpeza e qualidade de energia. O tempo de resposta de todo o sistema, deve ser suficientemente rápido para serem úteis para o controlo, enquanto que a velocidade necessária, deverá variar com a natureza do processo. Um computador de controlo industrial, tem vários atributos [35]:

- Tolerância ao ambiente fabril;
- Aprendizagem fácil de linguagens de programação;
- Operação facilmente monitorizada.

Em 1968, a GM Hydra-Matic (a divisão de transmissão automática da *General Motors*) emitiu um pedido de resposta para um substituto eletrónico, para sistemas de retransmissão *hard-wired*[1] com base em papel branco desenvolvido pelo Engº. Edward R. Clark. O vencedor da proposta foi Bedford. O primeiro PLC (controlador lógico programável), resultado desta proposta, foi chamado de "084", uma vez que, foi o octogésimo quarto projeto de *Bedford Associates*. [22]

Bedford Associates começou uma nova empresa dedicada ao desenvolvimento, fabricação, venda e manutenção deste novo serviço. O serviço chamava-se *Modicon* (COntrolador DIgital MOdular). Uma das pessoas que trabalhou neste projeto, foi Dick Morley, considerado por muitos como o "Pai"do PLC [11].

Antigamente, até meados da década de 1990, os PLCs eram programados utilizando painéis de programação próprios ou terminais de programação de uso especial, que muitas vezes tinham chaves de função dedicada, representando a variedade de elementos lógicos dos programas PLC [22].

[1] Utilizado para conectar (componentes eletrónicos, por exemplo) através de fios e cabos elétricos

A norma IEC 61131 representa uma combinação e continuação de diferentes padrões. Esta refere-se a 10 outras normas internacionais (IEC 50, IEC 559, IEC 617-12, IEC 617-13, IEC 848, ISO/AFNOR, ISO/IEC 646, ISO 8601, ISO 7185, ISO 7498), incluindo regras acerca de código de caracteres independentes, a definição da nomenclatura utilizada ou da estrutura de representações gráficas. Muitos esforços têm sido feitos desde o passado, para estabelecer um padrão da tecnologia de programação do PLC. A norma IEC 61131 foi a primeira que recebeu uma resposta positiva e uma boa aceitação internacional e industrial [20].

2.1.2 Controladores Lógicos Programáveis

O acto de implementar um sistema de controlo, para processos e maquinaria industrial, pode ser considerado como Automação. O seu objetivo principal é reduzir a necessidade de intervenção humana. Assim sendo, controlo e automação, andam normalmente de mão dada. As principais vantagens da automação podem ser resumidas na lista que se segue [40]

- Aumento da produtividade;

- Redução do custo associado ao tempo de funcionamento;

- Precisão no controlo;

- Rápida e antecipada deteção e notificação de ocorrência de falhas;

- Segurança na operação entre Homem e máquina.

Num sistema de automação onde é necessário um sistema de controlo de ações a realizar num dispositivo ou conjunto de dispositivos em ambientes industriais, com estados, operações lógicas e controlo de saídas, o controlador típico é o PLC (Controlador Lógico Programável). Trata-se de um dispositivo indicado para decisões e interpretações lógicas de ações a controlar um determinado processo. Tendo em conta a sua utilização sem a interferência do homem, ele é utilizado em automação.

Um controlador lógico programável (PLC) é um sistema eletrónico de funcionamento digital, que foi projetado para ser utilizado em ambientes industriais. Este sistema utiliza uma memória programável para armazenar as instruções dos utilizadores, pelas quais são implementadas através de funções especificas, tais como [18, 39]:

- Funções lógicas, sequenciais, temporais e aritméticas;
- Contadores;
- Entradas e saídas de controlo (*input/output*) analógicas.

A figura 2.1 ilustra um exemplo de um PLC.

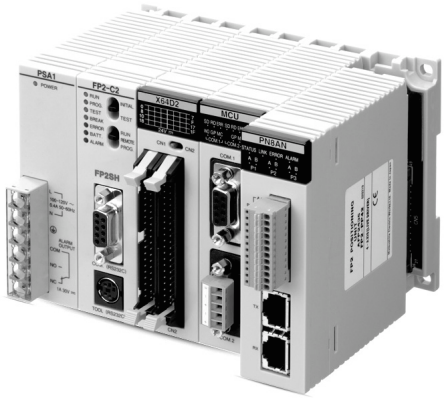


Figura 2.1: Exemplo de um PLC (retirado de [19]).

Um programa para PLC, normalmente, é executado repetidamente durante o tempo em que o sistema de controlo está a decorrer.

O estado das entradas físicas (*physical input points*) é copiado para uma área da memória acessível ao processador, muitas vezes chamado de "*I/O Image Table*"[15].

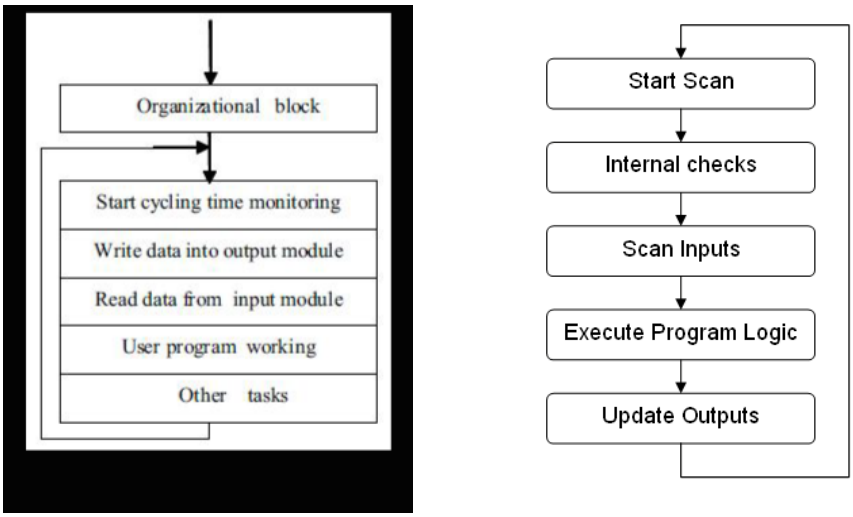
O programa é executado a partir da sua primeira instrução. Leva algum tempo ao processador do PLC avaliar todas as variações de estado e instruções implícitas, atualizando a tabela de imagens de entradas e saídas (*I/O Image Table*) com os estados das saídas (*outputs*). Este processo é normalmente referido como "*Cyclic Scan*"[13].

Um programa no CPU do PLC inclui um sistema operativo e um programa escrito por um utilizador. O sistema operativo é utilizado para lidar com tarefas (*tasks*), chamar um programa feito pelo utilizador, processar erros ou gerir áreas de armazenamento e comunicação. Um programa é elaborado pelo utilizador e utilizado para terminar as tarefas de controlo de automação.

Na figura 2.2a estão ilustrados os princípios fundamentais de um PLC.

Como pode ser visto na figura 2.2b, o PLC pode adotar um funcionamento com leituras dos sinais de entrada, execução lógica do programa, escrita das saídas e análise em modo cíclico (*scan mode*). As principais etapas referentes a este ciclo de funcionamento do PLC, estão resumidas nos próximos passos:

1. O sistema operativo inicia a monitorização em tempo cíclico;
2. É realizada a leitura interna dos sinais de entrada;
3. É executada a lógica do programa criado pelo utilizador;
4. São atualizados e escritos os sinais de saída;
5. No fim de todo o ciclo são implementadas todas as tarefas.



(a) Príncipio do PLC (retirado de [5]) (b) *Cyclic Scan* (retirado de [17])

Figura 2.2: Principais etapas de funcionamento de um PLC

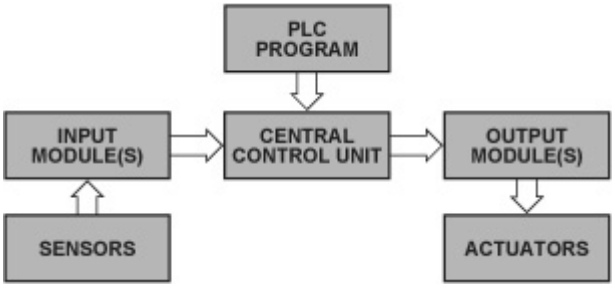


Figura 2.3: Sistema de alto nível de um PLC (retirado de [41])

Como podemos observar pela figura 2.3, existem cinco componentes básicos num sistema PLC, tais como o processador ou controlador PLC (unidade central de controlo), os módulos de entrada e saída (*I/O*) correspondentes aos sensores e atuadores respetivamente, o programa desenvolvido pelo utilizador, a fonte de alimentação e a estrutura física do autómato.

2.1.3 Programação de Autómatos

A norma IEC 31161-3 corresponde à terceira parte do padrão para controladores, ou seja, para linguagens de programação e tem o objetivo de auxiliar os engenheiros de controlo no desenvolvimento de aplicações utilizando uma ou várias linguagens de programação típicas de um PLC [1, 42].

A norma IEC 31161-3 abrange um conjunto de cinco tipos de linguagens de programação em autómatos: [42]

- Linguagem *Ladder* (LD);

- Linguagem SFC (*Squential Function Chart* ou baseada em *Grafcet*)
- Linguagem FBD (*Function Block Diagram*)
- Linguagem de Lista de Instruções (IL - *Instruction List*);
- Linguagem de Texto Estruturado (ST - *Structured Text*)

A linguagem LD (*Ladder*) é baseada em representações gráficas de lógica de relés, permitindo descrever uma sequência lógica de entrada de um processo, utilizando ligações, contactos e bobines. A linguagem FBD é também ela uma linguagem gráfica que expressa o fluxo de sinais e dados, utilizando uma combinação de funções e blocos de funções. A linguagem IL, é uma linguagem de programação de baixo nível, similar ao *assembley* em aplicações simples[38, 42].

A linguagem de Texto Estruturado é a linguagem utilizada como base de todo este projeto, sendo a linguagem utilizada para controlar o autómato (PLC) TSX3721 presente no laboratório utilizado. Como uma das cinco linguagens suportadas pela norma IEC 61131-3, é especializada e desenhada para programação de controladores lógicos programáveis, correspondendo a uma linguagem de alto nível, que está estruturada como um bloco e sintaticamente semelhante ao Pascal, no qual de baseia.

A parte do "estruturado", refere-se à programação de alto nível, e o "texto", à habilidade de utilizar texto em vez de símbolos como o *Ladder*, o FB ou SFC. Os vários tipos de linguagens de programação padronizadas pela norma IEC 61131-3 podem ser chamadas em blocos de funções por cada um dos diferentes tipos de linguagem.

Podemos afirmar que a linguagem de texto estruturado é uma das maneiras mais rápidas e mais fáceis de criar programas em controladores, consistindo e proporcionando, um conjunto de ferramentas de utilização rápida para desenvolvimento de programas, contendo algumas vantagens, destacando-se a facilidade de programar complexas equações matemáticas [8].

2.2 Construção de Compiladores em Ambientes de Programação

A tecnologia envolvente a um compilador tem muitas utilizações importantes e impactos em várias áreas da ciência da computação. Um projeto de um compilador exige rigor e perseverança quando aplicado a uma grande linguagem de programação.

2.2.1 Enquadramento Histórico

No final da década de 1950 foram propostas para máquinas várias linguagens de programação. O primeiro compilador foi escrito por Grace Hopper em 1952 para a linguagem de programação "*A-0*". O "*A-0*"funcionava mais como um carregador do que a noção moderna de um compilador [4].

O primeiro *autocode* e respetivo compilador, foram desenvolvidos pela Alick Glennie em 1952 para o computador *"Mark 1"* na Universidade de *Manchester*, sendo considerado por muitos como o primeiro compilador de linguagem de programação. [21]

Um *autocode* é uma linguagem de programação de alto nível. É o nome da família de sistemas de codificação simplificados (*simplified coding systems*), chamados mais tarde de linguagens de programação [6].

FORTRAN e COBOL, foram dois compiladores criados em 1957 e 1960 respetivamente. O FORTRAN era liderado por John Backus na IBM, sendo considerado como o primeiro compilador completo. Já o COBOL foi considerado uma linguagem prematura para se utilizar em múltiplas arquiteturas [4].

Para muitos domínios de aplicação, os compiladores eram utilizados cada mais em linguagens de programação de alto nível. Devido à funcionalidade em expansão, apoiada pelas mais recentes linguagens de programação e a crescente complexidade das arquiteturas de computador, os compiladores tornaram-se também eles mais complexos.

O primeiro interpretador (*interpreter*) de linguagem de alto nível foi provavelmente o *Lisp*. O *Lisp* foi implementado por Steve Russel num computador IBM 704, após ler um artigo escrito por John McCarthy [14].

O *Bison* é um descendente de yacc (*yet another compiler compiler*), um gerador de um analisador sintático (*parser*), criado entre 1975 e 1978 por Stephen C. Johnson na Bell Labs. Como o próprio nome afirma, a abreviação de "mais um compilador de compilador", indica que muitas pessoas estavam criando ou escrevendo *parsers* na altura. Richard Stallman da fundação FSF (*Free Software Foundation*), adaptou o trabalho de Corbett para utilizá-lo no projecto GNU[2], onde foi crescendo para aumentar um vasto número de novas funcionalidades, até evoluir na versão presente do *Bison*. O *Bison* é agora mantido como um projecto FSF e é distribuído como uma licença pública GNU [24].

Em 1975, Mike Lesk e Eric Schmidt escreveram o lex, um gerador de análise léxica. Viram-no como uma ferramenta autónoma e como um complemento e "companheiro"do yacc desenvolvido por Johnson. O lex também se tornou bastante popular, apesar de inicialmente ser relativamente lento e conter alguns *bugs*[3] [24].

[2]GNU é um sistema operativo de computador em Unix desenvolvido pelo *GNU Project*
[3]*Bugs*, em informática, são conhecidos como erros de um programa de computador

Schmidt, no entanto, passou a ter uma carreira bastante bem sucedida na indústria de computadores, onde chegou a ser CEO da Google até Abril de 2014.

Em 1987, Vern Paxson, do laboratório de Lawrence Berkeley, fez um versão escrita do lex em "ratfor"(uma versão Fortran prolongada) e traduziu-a para a linguagem C, chamando-a de *Flex* (*Fast Lexical Analyzer Generator*). Desde que ficou mais rápido e confiável que o lex AT&T, ultrapassou completamene o lex original, sendo agora um projeto *SourceForge*, sob a licença Berkeley [24].

2.2.2 Estrutura de um Compilador

O mundo inteiro necessita e depende das linguagens de programação, porque todo o *software* que corre nos nossos sistemas, em todos os computadores, está escrito em alguma linguagem de programação. O processo que permite um programa correr sem problemas, não é assim tão simples e direto, sendo necessário, ser traduzido numa forma e estrutura que possa ser executado por um computador.

O sistema de *software* que permite fazer uma translação e tradução de linguagens de programação, é o **compilador**. O compilador pode ler um programa numa linguagem (*source program*) e traduzi-la num programa equivalente noutra linguagem (*target language*), tendo como importante característica, o facto de conseguir reportar erros do programa fonte (*source*) que são detetados durante o processo de tradução (ver fig. 2.4) [4].

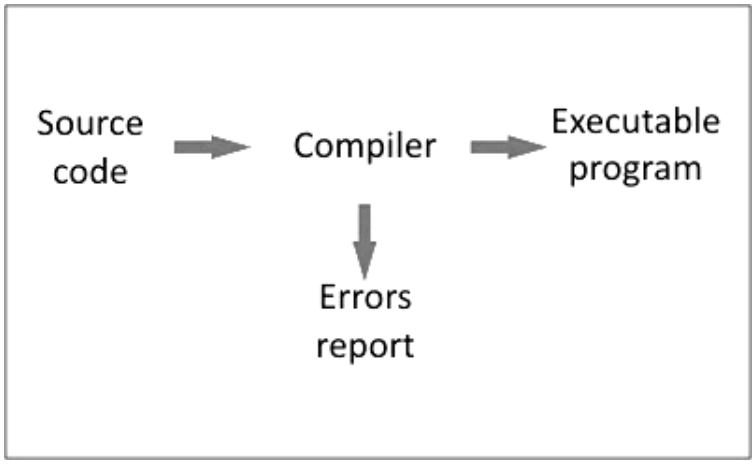


Figura 2.4: Um compilador (Retirado de [45]).

Um *interpreter* não é muito diferente de um compilador, convertendo também uma linguagem de alto nível. Cada vez que o *interpreter* recebe o código de linguagem de alto

nível a ser executado, converte-o num outro código intermediário antes do código da máquina. O código é executado separadamente numa sequência, encontrando, se for o caso, erros associados na sua interpretação. Sempre que encontra um erro, pára a interpretação e não traduz o próximo conjunto de códigos [44].

As principais diferenças entre um compilador (*compiler*) e um interpretador (*interpreter*) estão resumidas na próxima Tabela (2.1):

Tabela 2.1: Principais diferenças entre compiladores e interpretadores (*compilers/interpreters*) (Adaptado [44]).

Interpretador (*Interpreter*)	**Compilador (*Compiler*)**
O *interpreter* agarra num conjunto de instruções (*statement*), traduz-lo, executa-o e por fim agarra outro conjunto.	O *compiler* traduz o programa inteiro num outro e então, executa-o.
O *interpreter* vai parar a tradução depois de encontrar o primeiro erro..	O compilador gera um relatório de erro depois da tradução do programa inteiro
O *interpreter* leva muito menos tempo a analisar o processo do código de linguagem de alto nível.	Para o mesmo processo, o compilador leva muito mais tempo a realizá-lo.

Colocando de lado o tempo de processamento e análise, o tempo geral de toda a execução do código, é mais rápida no compilador do que no interpretador.

Até agora foi considerado que o compilador era uma caixa que mapeava um programa (*source*) num outro programa alvo (*target*) equivalente semanticamente. Especificando um pouco mais detalhadamente, o compilador é dividido em duas grandes parte para este mapeamento:

→ Análise (*analysis*);

→ Síntese (*synthesis*).

A parte responsável pela **análise** fragmenta o programa *source* em vários bocados e impõe uma estrutura gramatical, criando uma representação intermediária deste programa. Caso o programa de origem fornecido pelo utilizador (*source program*), contenha uma estrutura sintaticamente mal formada ou semanticamente sem sentido, então a parte de análise informa o utilizador com mensagens informativas. Uma das funções desta parte é juntar a informação acerca do programa de origem e armazená-la numa estrutura de dados chamada *symbol table* (tabela de símbolos) [4].

A secção da **síntese** constrói o programa alvo (*target*) desejado, através da representação intermediária e da informação da tabela de símbolos. Outra maneira pelo que a parte de análise (*analysis*) e a parte da síntese (*synthesis*) são chamadas muitas vezes é de

"*front end*" e "*back end*" respetivamente. O processo de compilação opera numa sequência de várias etapas ou fases (*phases*), transformando uma representação do programa *source* por outro. A tabela de símbolos (*symbol table*) vai armazenar informação referente a todo o código *source* e é utilizado por todas as fases (*phases*) do compilador [4].

Na figura 2.5 são mostradas as várias fases de um compilador.

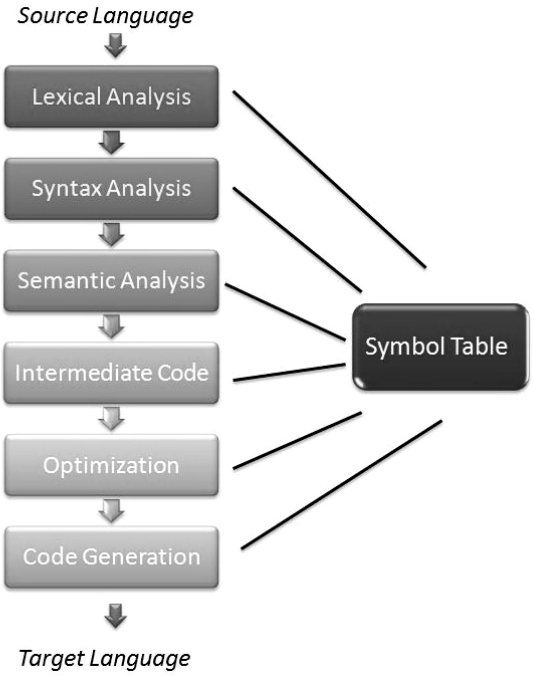


Figura 2.5: Fases de um compilador (Retirado de [10])

Como se pode ver pela Figura 2.5, a primeira fase do compilador corresponde à **análise léxica** (*lexical analysis or scanning*), em que o analisador lê uma sequência de caracteres e converte-os numa sequência de ***tokens*** (i.e. uma cadeia de caracteres significativos, símbolos léxicos ou simplesmente símbolos que fornece significado ao texto). A análise léxica funciona como um verificador de um alfabeto, inventado pelo utilizador ou não, em que se verifica se determinado caracter existe ou não nesse alfabeto[4].

A segunda fase do compilador é a **análise sintática** (*syntax analysis or parsing*). O *parser* utiliza os *tokens* produzidos pelo analisador léxico para criar tipos de representações intermediárias de árvores (árvores de derivação), que definem a estrutura gramatical das sequências de *tokens* [4].

Estas árvores de derivação (*parse trees*) são a representação gráfica de uma derivação, que mostra a estrutura hierárquica do programa (*source program*) [43].

A Figura 2.6 ilustra o processo e as etapas desde o programa *source* até à estrutura sintática.

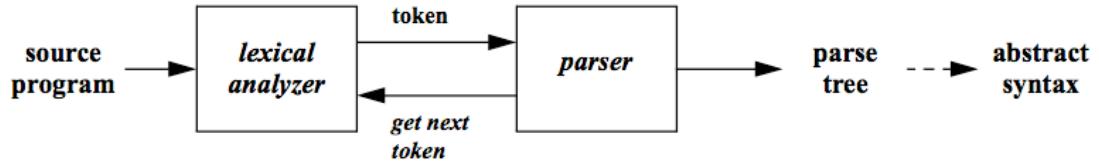


Figura 2.6: Processo de construção da estrutura sintática (retirado de [43]).

A **análise semântica** utiliza a árvore sintática e a informação da tabela de símbolos para verificar a consistência semântica das definições e a estrutura da linguagem do programa *source*, reunindo informações do tipo de dados, para posterior utilização na geração intermediária de código. Uma das partes mais importantes desta análise é a verificação de correspondência de cada operando para com os respetivos operadores [4].

Em processos de tradução de programas *source* para programas alvo, os compiladores devem construir pelo menos uma representação intermediária, como árvores de derivação e respetivamente árvores sintáticas, após a análise léxica (através do *lexical analyzer*) e análise sintática (através do *parser*). A criação de código numa linguagem alvo, é realizada através da entrada duma representação intermediária da linguagem do programa de origem (*source*) no *code generator*, saindo a linguagem alvo desejada (*target*) [4].

2.2.3 Lex e Yacc (*Flex* e *Bison*)

Conforme visto na secção 2.2.2, as principais fases de um processo de compilação, passa pela análises léxica, sintática e semântica (ver fig. 2.5, pág. 17).

O Lex e Yacc, ou versões melhoradas e mais rápidas *Flex* e *Bison* respetivamente, são os *softwares* correspondentes que permitem gerar estes vários tipos de analisadores em linguagens de programação.

O *Flex* é portanto uma ferramenta para gerar *scanners* (analisadores léxicos), que reconhecem padrões léxicos num determinado texto, lendo uma entrada (ficheiros de entrada ou *input* convencional), obtendo uma descrição do *scanner* a gerar [36].

Esta descrição é composta por regras, que constituem pares de expressões regulares e código C.

As expressões regulares são uma sequência de caracteres que formam um padrão de busca, utilizado para correspondência de *strings*, ou pares de *strings*, ou seja, como uma operação de "busca e reposição"[29].

O *Flex* após definir as regras léxicas com as respetivas expressões regulares, dividindo um ficheiro de entrada em peças, retorna um símbolo léxico por cada expressão encontrada, ou seja, uma etiqueta com um identificador associado, e o respetivo valor, chamado

de *token*. A Listagem de código 2.1 apresenta um exemplo de uma expressão regular e respetivo *token* associado, neste caso a definição de um *token* correspondente a um simples dígito.

```
[0-9]           return DIGITO;
```

Listagem 2.1: Exemplo de uma expressão regular e respetivo "token"associado.

O *Bison* vai fazer a segunda parte do trabalho, sendo responsável pela análise sintática e semântica, dentro das etapas definidas por um compilador. O *Bison* ao receber os *tokens* vindos do *Flex*, agrupa-os logicamente, e define toda a gramática predefinida pelo utilizador, através de um ficheiro yacc com as respetivas regras. As regras de produção criadas, vão levar à criação de ações, cuja implementação, normalmente é definida por funções programadas em linguagem C [24].

São as ações implementadas, que especificam o significado e a semântica das regras gramaticais definidas pelo ficheiro de entrada yacc. A Listagem 2.2 apresenta um pequeno conjunto de aplicação de regras de produção.

```
DIGITO '+' DIGITO               {$$=$1+$3;}
DIGITO '-' DIGITO               {$$=$1-$3;}
DIGITO '*' DIGITO               {$$=$1*$3;}
'(' DIGITO ')'                  {$$=$2;}
```

Listagem 2.2: Exemplo de aplicação de regras de produção.

Através da geração de código intermédio, os códigos Lex e Yacc gerados, são compilados e interligados por bibliotecas comuns, dando origem a um ficheiro executável capaz de reconhecer regras léxicas, e interpretar as respetivas ações, definidas pelas regras gramaticais e de produção. Este ficheiro é normalmente designado de reconhecedor (*parser*).

Toda a caracterização importante e essencial para o funcionamento e objetivos propostos pelos sistemas *Flex/Bison* (Lex/Yacc) estão especificados na secção 3.2.4.

A figura 2.7 ilustra a relação entre os programas de Lex (ou *Flex*) e Yacc (ou *bison*) respetivamente.

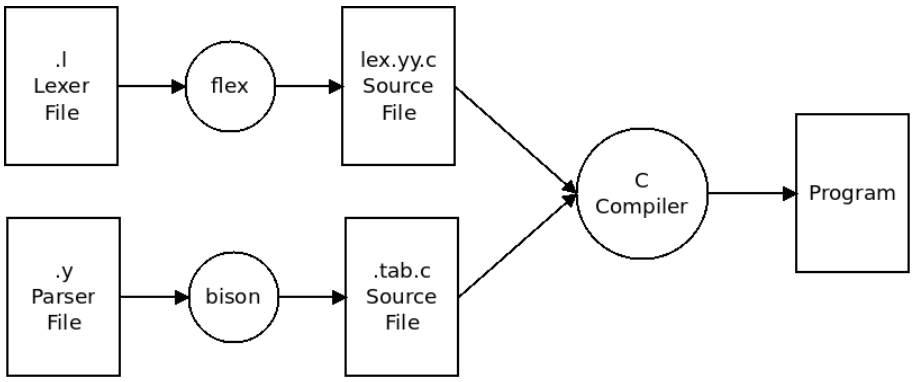


Figura 2.7: Processo geral de compilação *Flex/Bison* (retirado de [46]).

2.3 Projetos de Estudo

Como complemento ao trabalho desenvolvido neste projeto, foi importante o estudo e análise de alguns trabalhos similares aos realizados nesta dissertação, mas que apesar de apresentarem estruturas e metodologias parecidas, o objetivo proposto foi diferente, contribuindo assim para a aquisição de novos e importantes conhecimentos. São apresentados, de seguida, os principais aspetos abordados pelos projetos de estudo.

Em 2010, Jawad Hassan, num projeto de tese para a Universidade Linnaeus (Suécia) ([16]), desenvolveu um compilador capaz de compilar programas baseados em linguagem de Texto Estruturado e conseguir gerar uma estrutura de circuito digital equivalente aos programas originais. Os circuitos digitais foram apresentados em documentos XML, cujos esquemas para as estruturas dos circuitos digitais discutidos no projeto, foram desenvolvidos pela *Sauer-Danfoss*, uma das empresas fabricantes de controladores lógicos programáveis (PLCs).

O compilador em discussão foi desenvolvido baseando-se na linguagem C (tal como este projeto de tese desenvolvido para a criação do simulador de Texto Estruturado) e ANTLR3 como gerador de um analisador (*parser*). A especificação da linguagem de Texto Estruturado foi também traduzida pela forma estrutural *Backus Naur Form (BNF)* e o padrão visitante é adotado para implementar a análise semântica e a geração de código.

Os resultados obtidos incluem um compilador que traduz como fonte código de Texto Estruturado para XML. A exatidão, correção e desempenho da tradução dos programas de Texto Estruturado, são avaliados utilizando um esquema fornecido pela *Sauer-Danfoss*.

F. Narciso juntamente com alguns colaboradores, publicaram um artigo ([31]) intitulado "*A Syntactic Specification for the Programming Languages of the IEC 61131-3 Standard*",

no qual abordam a importância da definição da CFG (*context-free grammar*) para o desenvolvimento de um tradutor que pode gerar código de uma linguagem de programação de alto nível a partir de um programa escrito em qualquer uma das linguagens de programação descrita pela norma IEC 61131-3.

Tal importância reside no facto de que é possível a conceção e implementação de um analisador léxico, tendo em conta que a expressão regular que descreve uma linguagem de programação é um caso particular da CFG. Além disso, foi percebido que um analisador sintático pode ser concebido e implementado a partir desta gramática, que determina se uma dada instrução de programação pertence ou não às linguagens de programação geradas pelo CFG proposto.

Neste artigo foi ainda discutido e incentivado a proposta de utilização de normas abertas e de *open source* para a implementação do tradutor, representando uma contribuição importante, uma vez que, atualmente, a maioria dos tradutores para linguagens de programação para PLC, disponíveis no mercado, são desenvolvidos utilizando *software* licenciado, fechado e pago. O tradutor proposto no artigo, tem toda a generosidade de um *open source*.

Em 2010, João Martins e co-autores, no artigo ([27]), apresentaram uma nova abordagem de programas de controlo de teste em PLCs para automação de processos de ensino. Esta abordagem é baseada em linguagem do *software Matlab/Simulink*, no qual, o programa de controlo é traduzido num bloco de funções *Matlab* em ambiente *Simulink*, atuando sobre o modelo de um processo industrial durante todo o funcionamento da simulação. Foi desenvolvido um pacote de tradução, que traduz automaticamente o programa de controlo PLC, escrito em IL (*Instruction List*), em linguagem de *software Matlab/Simulink*.

Um conjunto de regras de tradução convertem a linguagem IL do PLC em linguagem *Matlab*, que através delas, o pacote de tradução produz um ficheiro (*m*-file). Este ficheiro é integrado no processo de simulação *Matlab/Simulink* como um bloco de funções chamado "`PLC Control Program`".

Em 2011, André Pereira realizou uma dissertação de mestrado ([37]), cujo trabalho desenvolvido propôs a possibilidade de análise, teste e validação de um programa de controlo imposto num PLC, permitindo a utilização de programas de controlo originais para controlar um modelo simulado de um processo industrial. A ideia base seria o próprio programa de controlo ser modelado, de modo a que o ambiente de simulação alvo o reconhecesse nativamente.

Este projeto ofereceu assim, um conjunto de 2 sistemas computacionais que permitem simular, em ambiente *Matlab/Simulink*, o controlo, através de um PLC, de um processo industrial modelado. Em particular, o controlo é programado em linguagem IL (*Instruction List*), apesar de ter a possibilidade de ser programado em outras linguagens proprietárias e expansíveis.

Foram desenvolvidos um conversor (`Matlaber`) e o tradutor (`UnifIL`). O primeiro recebe o código em linguagem IL e converte-o automaticamente num ficheiro compatível com o ambiente *Matlab/Simulink*, simulando o programa original, mas num ambiente *Simulink*. O tradutor normaliza o código PLC proprietário em código IL.

Tendo em conta que a tradução do código foi com recurso a dicionários de regras e tradução própria, então é também expansível a várias linguagens PLC de baixo nível, desde que um dicionário adequado seja criado.

CAPÍTULO 3

ARQUITETURAS PROPOSTAS, TECNOLOGIAS E IMPLEMENTAÇÃO

Este capítulo faz uma abordagem precisa e fundamentada acerca de toda a arquitetura utilizada, à tecnologia envolvente e à sua implementação prática.

O processo de implementação do projeto juntamente com toda a arquitetura envolvente utilizada é apresentada e estruturada na secção de metodologias propostas (3.1). A arquitetura utilizada (3.2) é dividida em três segmentos principais, especificando o esqueleto e a base da metodologia proposta anteriormente. Esta estrutura corresponde às três grandes etapas que constituem o projeto. A secção da implementação (3.3), descreve todo o desenvolvimento realizado e implementado ao longo do trabalho desenvolvido, e que contribuíram para se atingir os objetivos inicialmente propostos.

ARQUITETURAS PROPOSTAS, TECNOLOGIA E IMPLEMENTAÇÃO

3.1 Metodologia Proposta

O desenvolvimento de um sistema capaz de simular uma linguagem de programação, neste caso a linguagem de texto estruturado (ST), através de uma interface gráfica para utilizador e, a interpretação léxica, sintática e semântica do mesmo código, requer um conjunto de procedimentos e métodos a realizar.

A realização do projeto e dos seus respetivos passos envolve 3 etapas principais, sendo descritas de seguida. Como complemento e para melhor interpretação dos objetivos propostos é ilustrada a metodologia na Figura 3.1:

1) A validação de um código na linguagem desejada a simular (texto estruturado).

   - Para se obter um modelo de código em texto estruturado validado é necessário que o código funcione, em particular para esta tese, no autómato TSX 3721 da *Schneider* e que cumpra os requisitos propostos pelo *kit* utilizado no laboratório de estudo (SML - *washing machine*). Os testes do código serão realizados diretamente no autómato até os resultados serem os desejados e se obter um bom modelo de código de linguagem de texto estruturado. É pedido que cumpra com os requisitos propostos pelo *kit*, para que se tenha a certeza de que o código funciona bem.

2) Construção do compilador e interpretador de linguagem de texto estruturado.

   - Esta segunda etapa é provavelmente a etapa mais complexa e mais importante do projeto. Neste ponto, são criados dois ficheiros, um ficheiro que contém as regras léxicas da linguagem (*lexer file*), e outro ficheiro com as regras gramaticais (*parser file*). Utilizando duas ferramentas (*flex* e *bison*) são gerados um analisador léxico e um analisador sintático, respetivamente. Estes analisadores e as regras produzidas são compilados gerando um ficheiro executável, pronto a interpretar código de linguagem de texto estruturado. Para testes preliminares é possível correr o executável através da linha de comandos e testar código de texto estruturado, obtendo um resultado analisado e compilado. Caso não exista qualquer

erro léxico ou sintático são executadas as ações semânticas associadas às regras de produção utilizadas no reconhecimento do texto estruturado.

3) Criação da interface gráfica de utilizador.

- A construção do simulador envolve a utilização do programa de *Software Visual Studio*, por ser uma boa ferramenta para a criação de interfaces interativas para utilizadores (*GUI*). Tendo em conta, o facto do código utilizado na interface ser em C# é necessário a criação de uma DLL (*Dynamic Link Library*). Interpretando as regras léxicas e sintáticas previamente definidas, e compilando as funções definidas necessárias para a realização da interface gráfica (por meio do *Visual Studio*) é construído o simulador de texto estruturado. A validação dos resultados obtidos é conseguida utilizando e testando o código realizado e encontrado previamente como modelo a seguir no 1º passo (*ST validated code*) e verificando se se comporta como o modelo e *kit* SML utilizado no autómato TSX3721.

A arquitetura com a metodologia proposta encontra-se na Figura 3.1.

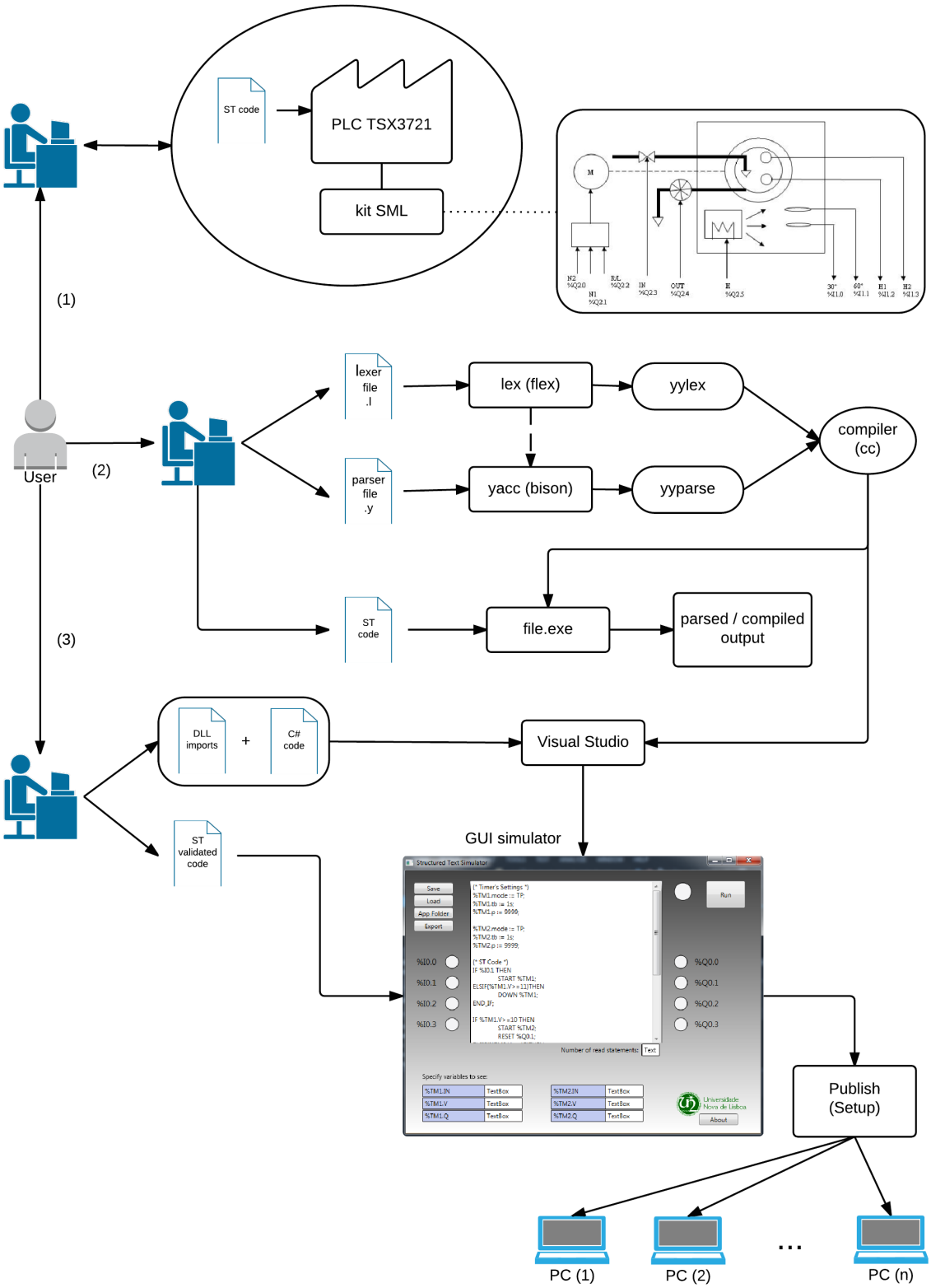


Figura 3.1: Arquitetura da metodologia proposta.

3.2 Arquiteturas e Tecnologias Utilizadas

A arquitetura utilizada para testes e simulações foi baseada no funcionamento do autómato TSX3721 juntamente com o KIT de simulação de uma Máquina de Lavar (kit SML). A linguagem utilizada para programar o autómato é o Texto Estruturado (ST: *Structured Text*). A realização da interface gráfica, que permite simular o comportamento do autómato com o *Kit* da máquina de lavar, foi baseada em C#.

3.2.1 Autómato TSX3721

O objetivo final do projeto é realizar um simulador de código de texto estruturado (`ST`) que funcione no controlador lógico programável *TSX Micro 3721* (Figura 3.2).

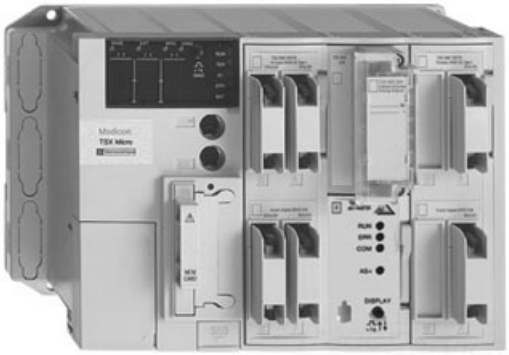


Figura 3.2: Autómato TSX3721(retirado de [2]).

Este controlador lógico é um autómato industrial programável compacto e modular, fabricado por uma empresa do grupo *Schneider Electric*. As suas principais funcionalidades são [34]:

- Contagem e controlo de posições analógicas / PID;
- Funções matemáticas;
- *Software* de aplicação multi-tarefas com funções activadas por um determinado evento.

O autómato TSX3721 apresenta a estrutura de um PLC comum industrial, representado por entradas (*Inputs*), saídas (*Outputs*), fonte de alimentação, e ligações com os restantes periféricos (PC, terminais, etc), como se pode observar na Figura 3.3.

O PLC TSX3721 possui algumas características técnicas em relação ao tipos de processador, como, o tipo *Bus* que pode ser *AS-interface, CANopen, Fipio*, com 1 *PCMCIA card*

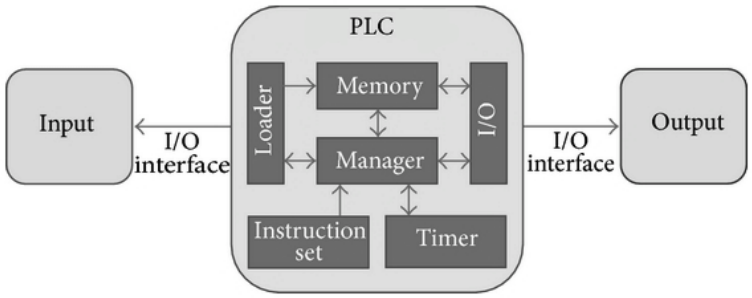


Figura 3.3: Estrura de um PLC industrial (retirado de [3])

cada, o tipo de *Network* (*Modbus Plus* e *Flipway* e um módulo externo para um rede *Ethernet TCP/IP*). A capacidade da memória pode ser integrada com 20k *words*, e 128k *words* + 128k *words* para armazenamento de ficheiros com extensão em PCMCIA. O PLC TSX3721 executa 200 ciclos em 1 segundo (*cycle time = 5 ms*)[12].

Alguns pontos característicos poderão ser vistos na Figura 3.4 com a respetiva legenda:

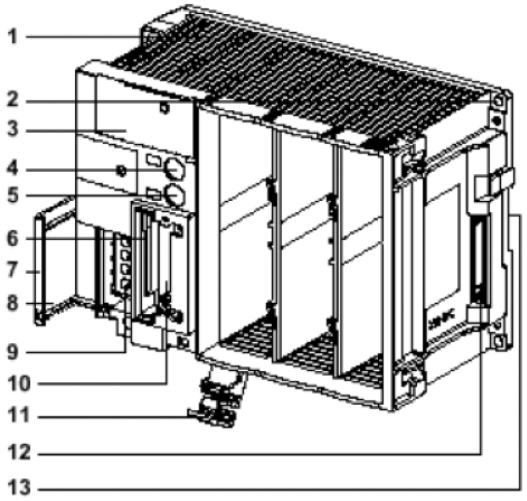


Figura 3.4: Pontos característicos do Autómato TSX3721 (retirado de [34])

1. Autómato (PLC)
2. Ponto de Montagem
3. Visor (*display*)
4. Terminal TER
5. Terminal AUX
6. Ranhura para extensão de memória
7. Tampa dos terminais da fonte de alimentação
8. Etiqueta
9. Terminais de alimentação
10. Ranhura para placa de comunicação *MODBUS* (tipo PCMCIA)
11. Tampa da bateria
12. Conector para módulo de extensão
13. Pontos de montagem DIN

O *hardware* disponível no laboratório consiste no PLC TSX3721 (Figura 3.4 - 1) e nos módulos entrada e saída (E/S). Este autómato utiliza dois módulos E/S principais e um módulo de extensão de memória. A comunicação utilizada no laboratório entre PC e PLC em estudo para a realização do projeto desenvolvido, foi realizada via porta série. Os dois módulos E/S correspondentes são [12]:

- Módulo TSX AMZ 600 (sinais analógicos de entrada e de saída, 0-10V e 4-20mA);
- Módulo TSX DMZ 28 DR (sinais discretos com valores [0; 24]V na entrada e relé na saída).

Tendo em conta a utilização de sinais discretos ao longo do projeto, o módulo utilizado para endereçamento de sinais de entrada e saída foi o Módulo TSX DMZ 28 DR. Este módulo tem 28 sinais de entrada e saída (28 E/S: *Input / Output* (I/O)), onde cada módulo pode ocupar as posições 1 a 10. A posição "0" está reservada para o CPU e para módulos integrados. O equipamento disponível no laboratório tem 16 entradas e 12 saídas (discretas).

O endereçamento de canais depende da localização geográfica do módulo no PLC. Na Figura 3.5 pode-se observar a localização do módulo DMZ28DR nas respetivas pos ições 1 e 2, e na Figura 3.6 é mostrado a estrutura de endereçamento de entradas e saídas.

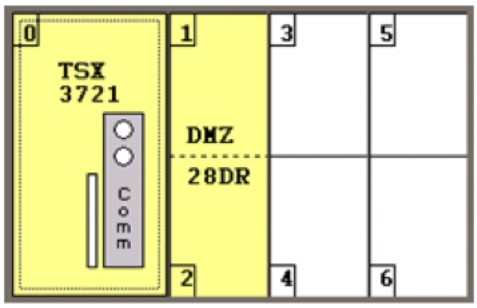


Figura 3.5: Localização do módulo DMZ28DR nas posições 1 e 2 [34].

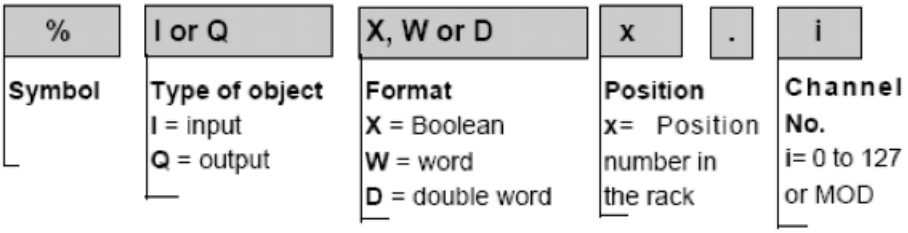


Figura 3.6: Endereçamento de entradas / saídas (I/O) [34].

Utilizando como exemplo as configurações ilustradas nas figuras anteriores, as entradas estão definidas no módulo 1 e as saídas no módulo 2. Analisando a estrutura descrita na Figura 3.6, as entradas e saídas podem ser referenciadas por:

- `%I1.x, x={0, 1, 2, ..., 15}`, por exemplo: `%I1.2`, significando a entrada 2 do módulo 1;
- `%Q2.x, x={0, 1, 2, ..., 11}`, por exemplo: `%Q2.3`, significando a saída 3 do módulo 2.

As configurações apresentadas são a base de atribuição e caracterização da programação utilizada mais tarde para este tipo de autómatos.

O PLC TSX3721 apresenta, como complemento, um visor (Figura 3.4 - 3), que indica o estado do PLC e dos sinais de entrada e saída E/S (I/O). Este visor é constituído por um conjunto de LEDs que correspondem exatamente a esses estados. Os principais LEDs são:

a) **RUN** (PLC a executar um programa);

b) **TER** (Transmissão de dados);

c) **I/O** (Falha nos I/O, Módulo desactivado, etc);

d) **ERR** (Falha no CPU);

e) **BAT** (Falha de bateria).

3.2.2 Processo a Controlar (kit SML "*Washing Machine*")

Para a realização de testes de algoritmos de controlo automático com os PLC's TSX3721, estavam disponíveis em laboratório os *kits* de simulação de semáforos (SSF) e de máquina de lavar (SML). O *kit* utilizado para testes de código ST em laboratório no autómato foi o *kit* SML ("*Washing Machine*" - Figura 3.7).

O Módulo SML é utilizado para simular o controlo e monitorização de uma máquina de lavar. Permite também o controlo de um programa com interruptor para ligar / desligar, ciclos de lavagem longos ou padronizados e ajuste de temperatura. Processos tais como lavagem e centrifugação (2 velocidades), aquecimento da água (*On/Off*), entrada de água e drenagem, podem ser assim, simulados. Este *kit* permite ainda testar a realimentação de sinal após a entrada de água, após se atingir vários níveis de água, ou ainda, atingir várias temperaturas após um período de aquecimento durante a simulação.

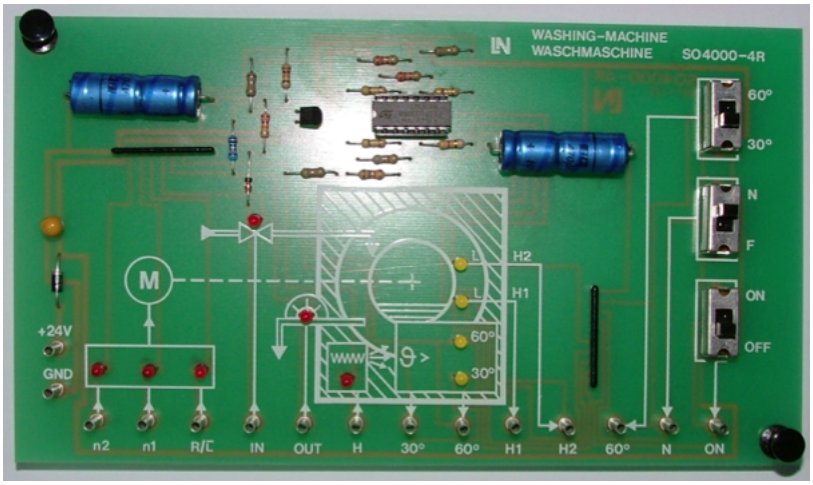


Figura 3.7: *Kit* simulador de máquina de lavar (SML - "*Washing Machine*") [25].

Este *kit* de simulação possui algumas características técnicas que são apresentadas de seguida [25]:

Inputs: 5 entradas digitais;

Outputs: 7 saídas digitais;

Suporte de Alimentação: 24V DC / 100mA;

Ligação: Conexão de 2mm via socket;

Dimensões: 110 x 195mm (HxW);

Peso: 0.1Kg.

Analogamente ao que foi especificado no secção 3.2.1 sobre as configurações das entradas e saídas (Figura 3.6), os *inputs* são referenciados por `%I1.x`, com x={0, 1, 2, ..,6}, situando-se todos no módulo 1, e os *outputs* são referenciados por `%Q2.x`, com x={0, 1, 2, ..,5}, situando-se estes no módulo 2 do DMZ28DR (Figura 3.5).

Os endereços de ligação ao PLC TSX3721 do *kit* SML são mostrados na Tabela 3.1.

Tendo em conta o número de endereços de entrada e saída definidos anteriormente é fácil de perceber as inúmeras opções e situações diferentes que podemos simular no *Kit* SML. Para se evitar ambiguidades e se ter um modelo de projeto coerente e que siga uma linha de pensamento constante, optou-se por arranjar especificações de funcionamento para se poder simular e chegar a um mesmo objetivo final, durante a realização de todas as fases de construção do projeto. A Figura 3.8 ilustra o mesmo modelo SML anterior, representando melhor as especificações de funcionamento propostas.

Tabela 3.1: Endereços de Ligação ao PLC TSX3721.

Entradas		**Saídas**	
%I1.0	Termóstato T30º	**%Q2.0**	Acção N2 (comando de motor de rotação, veloc v2)
%I1.1	Termóstato T60º	**%Q2.1**	Acção N1 (comando de motor, veloc v1)
%I1.2	Sensor de nível H1	**%Q2.2**	Acção R/L (Ligação estrela-triângulo)
%I1.3	Sensor de nível H2	**%Q2.3**	Acção IN (Abrir electro-válvula de admissão de água)
%I1.4	Botão 30º/60º (0 → 30º, 1 → 60º)	**%Q2.4**	Acção OUT, ligar o motor de extração de água
%I1.5	Botão N/F (1 → nível H!, 0 → nível H2)	**%Q2.5**	Acção H, aquecimento da água
%I1.6	Botão ON/OFF (1/0)	-	-

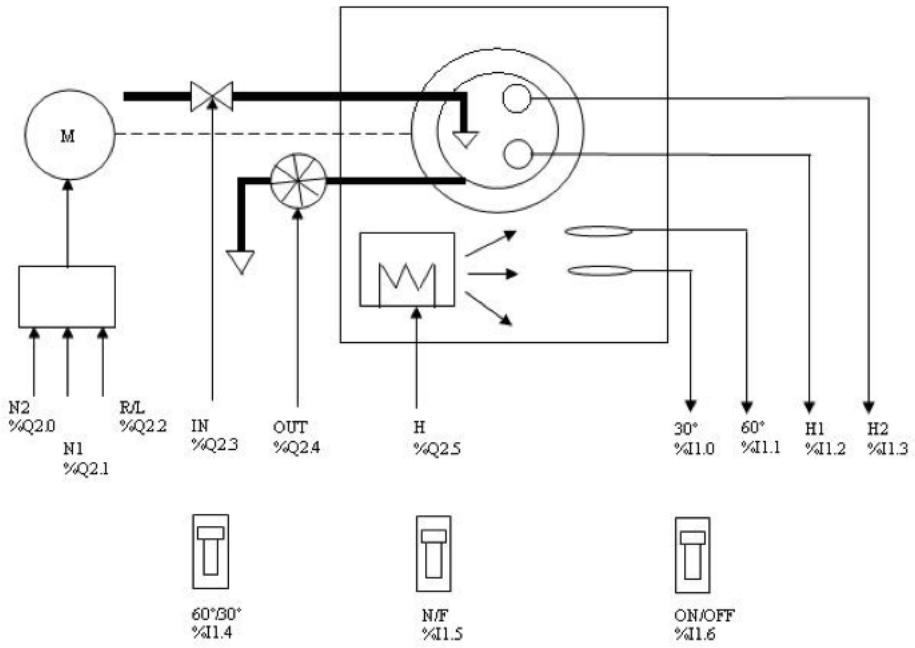


Figura 3.8: Funcionamento do *Kit* simulador de máquina de lavar (SML - "*Washing Machine*").

As especificações de funcionamento tomadas como exemplo a seguir para o desenvolvimento do projeto, particularmente para o desenvolvimento do código ST (*Structured Text*), são:

(1º) O botão (`%I1.6`) deverá ser o botão de impulso de marcha;

(2º) O botão (`%I1.5`) deverá ser o botão de impulso de paragem;

(3º) Após o arranque da máquina, a cuba deverá encher até ao nível "H2", por ação de "IN" (`%Q2.3`);

(4º) Logo de seguida, o motor "M" deve rodar por ação de "N2" (`%Q2.0`), durante 10s;

(5º) Por fim, a cuba deverá esvaziar por ação de "OUT" (`%Q2.4`) durante 15s.

3.2.3 Caracterização do Código ST (Texto Estruturado)

Conforme introduzido na secção 2.1.3, a linguagem de texto estruturado é uma linguagem para sistemas complexos, sendo uma linguagem estruturada de alto nível desenhada e projetada para processos de automação, fazendo parte do padrão IEC 61131-3.

Tendo em conta os requisitos e objetivos impostos a realizar no kit SML (ver Figura 3.8) é importante exemplificar e especificar as principais características da linguagem de texto estruturado, apresentando nesta secção um conjunto de regras, comportamentos e estrutura da linguagem.

As principais instruções da linguagem (ST) podem ser resumidas na lista que se segue:

- Instruções do tipo*bit*;
- Instruções aritméticas e lógicas em *words* ou *double words*;
- Instruções aritméticas com *floating points*;
- Comparações numéricas entre *words*, *double words* e *floating points*;
- Conversões numéricas;
- Instruções sobre tabelas de *bits*, *words*, *double words* e *floating points*;
- Instruções com *strings* de caracteres;
- Comparações alfanuméricas;
- Instruções com temporizadores (*timers*);
- Instruções de controlo;
- Instruções com blocos de funções;
- Instruções com trocas explícitas;
- Instruções com aplicações específicas (comunicação, controlo PID, etc).

O endereçamento funciona exatamente como foi exemplificado na secção 3.2.1, utilizando a configuração de endereçamento de entradas e saídas do autómato TSX3721. Relembrando a Figura 3.6, define-se os principais caracteres relativos aos objetos *bit* e *main word* nos módulos I/O, como:

[**Tipo do objeto**]

I e *Q*: Os módulos físicos dos *inputs* e *outputs*, trocam esta informação implícita a cada leitura da tarefa para a qual são anexados. Outros tipos de dados (*status, command words*, etc) podem também ser trocados, se solicitados pela aplicação.

[**Formato** (*Size*)]

Para objectos em formato booleano, o X pode ser omitido.

[**Número e Posição do Canal**]

A base modular do autómato TSX3721 utilizado é caracterizada pelas posições 1 e 2.

A linguagem de programação ST é baseada em escrita de caracteres alfanuméricos, onde uma instrução (*statement*) constitui a unidade básica da linguagem estruturada, e uma série de instruções (*statements*) são utilizadas para definir um programa.

O programa principal pode ser denominado por *main* e definido entre as declarações de `"PROGRAM"` e `"END_PROGRAM"`. Pode-se considerar que um programa ST é uma lista de *statements*, onde cada *statement* termina com um separador ponto e vírgula (";").

A linguagem de texto estruturado não é sensível às maiúsculas ou minúsculas mas é importante ou útil, que se utilize as variáveis como minúsculas e as instruções como maiúsculas. Indentar e comentar bem, também é fundamental para a legibilidade do programa.

Os nomes utilizados como código fonte principal (identificadores de variáveis, constantes, palavras-chave da linguagem, etc), são separados com separadores inactivos (caractere espaço, *end of line* ou tabulações) ou separadores activos, que têm um significado bem definido (exemplo: o separador ">" indica uma comparação "maior que"). Os comentários são definidos entre `(*...*)`.

Alguns tipos básicos de instruções em ST (*statements*) são exemplificados de seguida:

- Instrução de atribuição (variável `:=` expressão)
- Instruções de selecção (`IF, THEN, ELSE, CASE...`)
- Instruções de iteração (`FOR, WHILE, REPEAT,...`)
- Instruções de controlo (`RETURN, EXIT,...`)

Para melhor interpretação da estrutura e caracterização da linguagem de texto estruturado, serão apresentados alguns exemplos.

Como todas as linguagens de programação, as funções lógicas (`AND, OR`, ...) são um requisito obrigatório. Como foi visto, a atribuição de uma variável é realizada utilizando a instrução "`:=`".

Exemplos de atribuição de variáveis e utilização de funções lógicas são mostrados na Listagem 3.1.

```
%M1 := %I1.0 AND %I1.1;
%Q2.1 := %M1;

%Q2.2 := %I1.0 OR %I1.1;
%Q2.3 := %I1.0 AND NOT %I1.1;
%Q2.4 := (%I1.0 AND NOT %I1.1) OR %I1.2;
```

Listagem 3.1: Exemplo de Linguagem ST com Funções Lógicas.

Neste exemplo, a variável de memória binária `%M1` serve para guardar ou lhe ser atribuída o valor lógico da instrução [`%I1.0 AND %I1.1`]. A variável de saída `%Q2.0` vai conter esse mesmo valor que `%M1`. As outras instruções são outros exemplos de utilização de funções lógicas.

Outra instrução útil e comum no mundo da programação e obviamente em programação na automação é a utilização de ciclos.

Dentro das estruturas de controlo, existem quatro tipos principais:

> A ação repetitiva `FOR`;

> As ações iterativas condicionais `WHILE` e `REPEAT`;

> A ação condicional `IF`.

Cada estrutura de controlo é fechada entre palavras chave, começando e terminando na mesma instrução. É possível agrupar estruturas de controlo umas dentro de outras, independentemente do seu tipo, e serem precedidas ou seguidas por outra qualquer linguagem de instrução. Uma condição pode ser múltipla e a ação representa uma lista de instruções.

Considerando um programa na Listagem 3.2 que encontre a média de dez valores num vetor de reais "`v[]`", o ciclo `FOR` no exemplo, vai percorrer o ciclo dez vezes adicionando os valores do vetor. No fim será feita a média do somatório.

Podemos observar o exemplo (Listagem 3.3), agora utilizando o ciclo WHILE. As principais diferenças estão presentes na inicialização do valor e respectiva actualização manual do iterador "`i`".

```
sum := 0;
FOR (i := 0 TO 9) DO
        sum := sum + v[i];
END_FOR;
media := sum / 10;
```

Listagem 3.2: Exemplo de programa para média de 10 valores em memória com um ciclo FOR.

```
sum := 0;
i := 0;
WHILE(i<10) DO
        sum := sum + v[i];
        i := i + 1;
END_WHILE;
media := sum / 10;
```

Listagem 3.3: Exemplo de programa para média de 10 valores em memória com um ciclo WHILE.

A instrução iterativa condicional `REPEAT` tem um objectivo semelhante aos exemplos apresentados, mas executa uma ação repetitiva até a condição ser verificada. Ao contrário do que acontece nos dois outros tipos de ciclos, a condição só é testada após a ação ter sido executada. Se a condição for primeiro avaliada, o seu valor é "`FALSE`" e a ação é executada uma outra vez.

Um dos tipos de instruções mais utilizados e mais importantes em linguagem de texto estruturado (ST) é a ação condicional `IF`. A instrução efetua uma ação se a condição for verdadeira.

A Figura 3.9 retrata exatamente o comportamento deste tipo de condicionais.

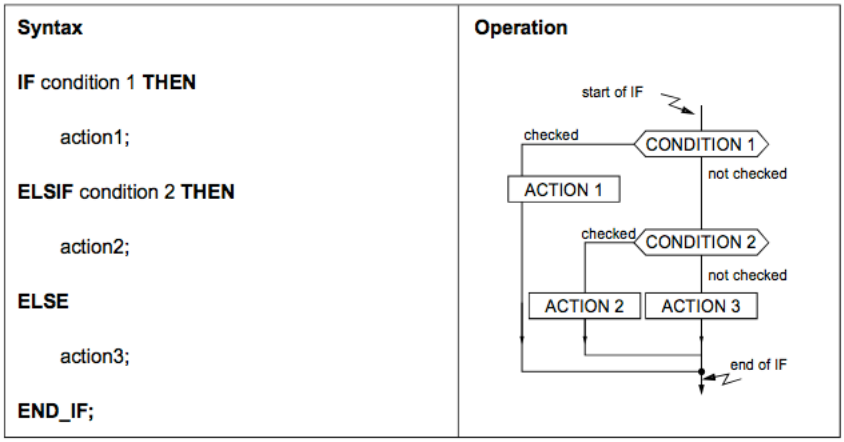


Figura 3.9: Forma geral da instrução condicional IF.

Como se pode observar pela sintaxe apresentada, caso a ação 1 e ação 2 não sejam realizáveis, então será executada a ação 3. É necessário ter em conta algumas características

em relação à estrutura das condições:

- Pode haver múltiplas condições;
- Cada ação representa uma lista de instruções;
- Algumas estruturas de controlo "`IF`" podem ser aglomeradas;
- Não há restrição para o número de instruções "`ELSIF`";
- Apenas uma instrução "`ELSE`" é permitida para cada "*IF statement*".

Considerando um programa simples que apenas ligue e desligue um motor (`%Q2.2`), algumas instruções com condições, são exemplificadas na Listagem 3.4.

Este exemplo está dividido em duas partes, uma com uma condição `IF` normal, onde se utiliza a negação `NOT` como condição para ver se a variável auxiliar de memória `%M1` está activa ou não, e caso não esteja é colocada com o valor lógico "1" através da instrução `SET`. A segunda parte realiza duas condições (`IF` e `ELSIF`), verificando se os *inputs* `%I1.2` e `%I1.3` respectivamente, estão activos, realizando correspondentemente as suas instruções. Note-se que a instrução `RESET` coloca de novo a variável com o valor lógico "0".

```
IF NOT (%M1) THEN SET %M1;
END_IF;

IF (%I1.2) THEN
        SET %Q2.2;
ELSIF (%I1.3) THEN
        RESET %Q2.2;
        RESET %M1;
END_IF;
```

Listagem 3.4: Exemplo de programa ligar/desligar motor (condicional IF).

Tendo em conta os objectivos propostos à realização do exemplo no kit SML (ver Figura 3.8) a utilização de temporizadores (*Timers*) é uma ferramenta chave e crucial.

Em linguagem ST ou em automação, os temporizadores (*timers*) têm três modos de operação:

- **TP**: é utilizado para criar um pulso com duração precisa;
- **TON**: é utilizado para gerir atrasos no arranque ou envolvimento (*on delays*);
- **TOF**: é utilizado para gerir atrasos no desligar ou libertação (*off delays*).

A representação gráfica que se segue (Figura 3.10), exibe um bloco de uma função *timer*:

Os atrasos ou os períodos dos pulsos são programáveis e podem, ou não, ser modificados através do terminal.

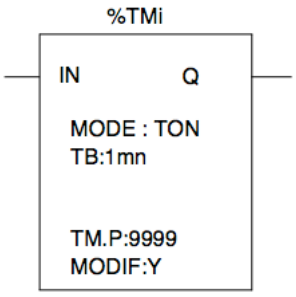


Figura 3.10: Bloco de uma função de temporizador (*Timer TON*) [9].

Existem algumas maneiras de encarar a programação ou a utilização do temporizador (*timer*) em linguagem ST. Uma maneira prática de se utilizar temporizadores é simular atrasos temporais para alguma ação, ou delimitar muito bem a duração de algum acontecimento.

O primeiro caso que se segue no exemplo mostrado na Listagem 3.5, simula o funcionamento de um motor durante 10s. Para se definir este tempo de duração é necessário configurar o tipo de *timer* a utilizar (TON, TOF, TP), o valor predefinido da duração (%TMi.P), e a escala temporal ou base de tempo (TB). As configurações utilizadas no exemplo (Listagem 3.5) são:

[TP]: O seu *bit* de *output* (%TMi.Q) é definido a 1 desde a ordem de inicio (`START`), até os 10 segundos passarem, neste caso. (%TMi.V=%TMi.P);

[%TMi.P]: Vai conter o valor 10;

[TB]: Vai conter uma base temporal de 1s.

Como se pode verificar pela Listagem 3.5, o *input* `%I1.0` corresponde ao botão de arranque, afectando a variável auxiliar de memória `%M1`. Se esta variável estiver definida com o valor lógico "1", então o *timer* é iniciado (`START`), e o motor (`%Q2.0`) durante o tempo predefinido nas configurações (10s), vai estar a trabalhar (`%Q2.0:=%%TM1.Q;`). Assim que o valor corrente do temporizador (`%TM1.V`) atingir o máximo estipulado (`%TM1.V:=%%TM1.P;`), então, o temporizador é desligado e a a memória auxiliar é redefinida a "0".

Outra maneira de programar um temporizador é jogar com as passagens de estado que as instruções `START` e `DOWN` criam. A instrução `START`, gera uma transição de estado de "0" para "1" (*rising edge*) na entrada do bloco do *timer*. A instrução `DOWN`, gera uma transição de "1" para "0" (*falling edge*), também na entrada do bloco do *timer*.

Na Listagem 3.6 é possível notar um caso aplicável deste tipo de utilização. Existe a verificação de mudança de estado do *input* `%I1.0` através do detetor de disparo `RE` (*rising edge*), iniciando o *timer* (`START%TM1;`) caso se verifique , e desligando o *timer*

```
IF (%I1.0) THEN
        SET %M1;
ELSE
        RESET %M1;
END_IF;

IF (%M1) THEN
        START %TM1; %Q2.0 := %TM1.Q;
        IF (%TM1.V := %TM1.P) THEN
                RESET %M1;
                DOWN %TM1;
        END_IF;
END_IF;
```

Listagem 3.5: Funcionamento de um motor (%Q2.0) durante um período temporal.

(`DOWN%TM1;`), caso a mudança de estado contrária de `I1.0` através do detetor de disparo `FE` (*falling edge*) se verifique. O funcionamento do motor (`%Q2.0`) é análogo ao exemplo anterior da Listagem 3.5, tal como todas as configurações técnicas do tipo de *timer*.

```
IF RE %I1.0 THEN
        START %TM1;
ELSIF FE %I1.0 THEN
        DOWN %TM1;
END_IF;
%Q2.0 := %TM1.Q;
```

Listagem 3.6: Utilização de transições de estado (RE/FE) no uso de temporizadores

Como linguagem de programação, a linguagem de texto estruturado (ST) é capaz de funcionalidades complexas e muito mais completas, dentro das funcionalidades revistas nesta secção. Conforme o objetivo proposto para o funcionamento com o *kit* SML, as principais instruções e características do texto estruturado (ST), foram apresentadas.

3.2.4 Caracterização do Lex / Yacc

Tendo em conta a construção de um compilador capaz de interpretar uma linguagem de programação, neste caso a linguagem de Texto Estruturado (ST), e gerar um executável que reconheça o código ST lido e proceda à respetiva interpretação, as ferramentas computacionais que melhor se adaptam a este tipo de sistemas, são sem dúvida, como já foi referido, o Lex e o Yacc.

O Lex é uma ferramenta, mais propriamente, um gerador de *scanners* ou analisadores léxicos. Um *scanner* é um programa que reconhece padrões léxicos em texto. O programa flex (*fast lexical analyzer*), moderna substituição do clássico lex, lê os ficheiros de entrada

indicados, ou a sua entrada padrão, caso não haja nenhum nome de ficheiro dado, para uma descrição de um *scanner* para gerar. Esta descrição está na forma de pares de expressões regulares e código C, chamado de regras. O flex c ria como *output*, um ficheiro fonte em C por definição, chamado "`lex.yy.c`", definindo uma rotina "`yylex()`". Este ficheiro ao ser compilado e acoplado com a biblioteca de execução do flex, pode produzir então, um executável. Ao correr, este executável analisa a sua entrada para ocorrências de possíveis expressões regulares. Caso encontre uma, então ele executa o correspondente código C [36].

O lex transforma as expressões e ações definidas pelo utilizador, especificadas num ficheiro fonte com as regras lexicais (podemos chamar *Source File*), numa linguagem de propósito geral interpretado pelo lex. A rotina gerada vai-se chamar "`yylex()`". Ao ser gerada, vai reconhecer as expressões que vão estar num ficheiro de entrada (*input File*), e vai executar as respetivas ações para cada expressão que for detetada. A visão geral deste funcionamento pode ser encontrada na Figura 3.11.

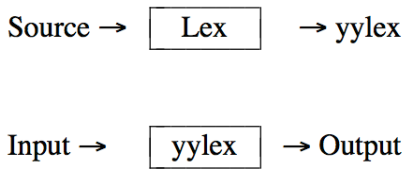


Figura 3.11: Visão geral da arquitetura Lex (retirado de [23]).

O formato geral do código fonte do lex é:

Definições
%%
Regras
%%
Sub-rotinas

A entrada do lex, ou seja, o código fonte (*source*) é divido em três secções com "%%" como divisores. O mínimo absoluto que se pode ter num programa em lex é só uma divisão com "%%", (sem definições nem regras), onde traduz para o programa a entrada numa saída inalterada. No formato geral dos programas em lex, as **regras** representam as decisões de controlo do utilizador, correspondendo a duas colunas, onde a coluna da esquerda contém as expressões regulares e a coluna da direita, contém as ações.

As **ações** (*actions*) são fragmentos do programa para serem executados quando as expressões são reconhecidas [23].

As **expressões regulares** podem conter caracteres de texto (que fazem ligar os caracteres correspondentes às *strings* as serem comparadas) ou até caracteres de operação (que especificam repetições, escolhas, e outras características). As letras do alfabeto e os dígitos, são sempre considerados caracteres de texto [23, 36].

Os ***tokens*** são representações numéricas de *strings* que simplificam o processo. Uma expressão regular especifica um conjunto de *strings* para serem correspondidas em *tokens*.

Na Tabela 3.2, podemos verificar alguns casos de primitivas padrão utilizadas em expressões regulares [23].

Tabela 3.2: Primitivas de correspondência padrão em expressões regulares.

Metacharacter	**Significado**
.	Qualquer caracter excepto a nova linha (*newline*)
\n	Nova linha (*newline*)
*	Zero ou mais cópias da expressão precedente
+	Um ou mais cópias da expressão precedente
?	Zero ou uma cópia da expressão precedente
^	Início da linha
$	Fim de linha (*end of line*)
a | b	**a** ou **b**
(ab)+	Um ou mais cópias de **ab** (*grouping*)
"a + b"	Literal **a+b**
[ ]	Classe de caracteres

Quando uma expressão escrita é correspondida, o Lex executa a ação correspondente. Sempre que o utilizador, utilizando o Lex, deseja absorver toda a entrada, sem produzir qualquer saída, então, deve fornecer regras para combinar com tudo. A ação padrão, como já foi referido, consiste em copiar a entrada para saída cujas as *strings* não são correspondidas. A situação mais normal e comum é o Lex ser utilizado com o Yacc. Neste caso, existe a construção de ações em vez de se copiar a entrada para a saída e, em geral, uma disposição que meramente copia normalmente é omitida.

A Tabela 3.3 apresenta alguns exemplos de expressões regulares e respetivas correspondências em sequências de caracteres [23].

Uma das coisas mais simples que se pode fazer para se ignorar o *input* é especificar em código C, a declaração de nulo (*null statement*). A regra frequente que, como ação, provoca este resultado é `"[\t\n]"`, que faz com que três caracteres de espaço (*blank, tab, newline*) sejam ignorados [23].

Tabela 3.3: Exemplos de aplicações padrão em expressões regulares.

Expressão	**Correspondência**
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
[a-z]	Qualquer letra de **a-z**
[-az]	Uma destas letras: **-, a, b**
[^ab]	nada exceto: **a, b**
[a ^b]	Um de: **a, ^, b**
[a | b]	Um de: **a, |, b**
a | b	**a** ou **b**
[\t \n]+	Espaço em branco
[A-Za-z0-9]+	Um ou mais caracteres alfanuméricos

Para se perceber melhor a utilização prática de expressões regulares e respetivas ações, vai ser mostrado um exemplo simples da aplicação numa calculadora. Este exemplo é dos mais comuns, porque a abordagem é bastante simples e percebe-se bem o seu funcionamento e a lógica envolvida. O código da Listagem 3.7 ilustra um exemplo de uma aplicação para uma calculadora. O código foi desenvolvido durante testes de estudo, antes da implementação principal do projeto da tese.

```
hex_digit               [0-9a-fA-F]
oct_digit               [0-7]

digit                   [0-9]
inteiro                 {digit}+
fraccionario            {inteiro}\.{inteiro}
int_frac                {inteiro}\.({inteiro})?
exponente               [eE][+-]?{inteiro}
real                    {int_frac}{exponente}?
%%
[ \t]   ;
{real}  { printf("\nLeu o numero"); yylval = atof(yytext);  return NUMERO;}
\n      { printf("\n");                                     return('\n');}
\+      { printf("\nDetectou uma Soma");                    return(SOMA);}
\-      { printf("\nDetectou uma Subtraccao");              return(SUBT);}
\*      { printf("\nDetectou uma multiplicacao");          return(VEZES);}
\/      { printf("\nDetectou uma Divisao");                return(DIVIDIR);}
%%
```

Listagem 3.7: Ficheiro *source* do Lex para um exemplo de uma calculadora

O excerto de código mostrado na Listagem 3.7 está dividido em duas partes principais da estrutura formal de um ficheiro lex (separadas pelos caracteres %%). A primeira parte, dirigida às definições, especifica um conjunto de expressões regulares que, com o acoplamento ou conjugação de umas com as outras, vai definindo representações de vários tipos de dados. Neste caso pode-se observar que um número real é um conjunto de números fracionários ou inteiros com ou sem exponente associado, que um número

fracionário é o conjunto de dois números inteiros separados por um ".", e que por sua vez um número inteiro é o conjunto de um ou mais dígitos. Nas definições foi definido, portanto, o que é um real através de expressões regulares.

Na parte das regras, vão ser associadas as verdadeiras ações do Lex, ou seja, para cada tipo de entrada reconhecido, vai ser executado uma instrução em código C que desempenhe a ação premeditada pelo utilizador. Neste exemplo, e porque vai haver uma ligação Lex e Yacc, todas as expressões reconhecidas e especificadas nas regras, são retornadas com o respetivo *token* associado. No código (Listagem3.7) é possível verificar alguns reconhecimentos de operações aritméticas e números, com os respetivos *tokens* retornados.

O lex pode ser utilizado sozinho para transformações simples em nível léxico, mas pode também ser utilizado com um gerador (*scanner*), para executar a fase de análises léxicas.

Quando o analisador léxico encontra identificadores nos fluxos de entrada, coloca-os numa Tabela de símbolos. A tabela de símbolos pode conter também, outra informação tal como o tipo de dados (inteiro ou real) e a localização de memória de cada variável. Todas as referências subjacentes aos identificadores, referem-se ao índice da tabela de símbolos apropriado [32].

Quando um *scanner flex* ou um analisador léxico, retorna um fluxo de *tokens*, cada *token* está dividido em duas partes, o número do *token* e o valor do *token*. No fundo, um *token* é um pequeno número. Todos estes números são arbitrários, exceto o número zero que está reservado para o "*end_of_line*". Quando o *bison* (versão moderna do Yacc) cria um *parser* (analisador sintático), atribui os números dos *tokens* automaticamente começando no 258 (evitando colisão com outros caracteres que acabam no 257), criando um "ficheiro.h" com as definições destes números. Um valor de *token* identifica a que grupo de *tokens* similares esse *token* pertence [24].

A variável "`yyval`" armazena o valor do *token*, ou seja, um inteiro. Mais tarde veremos que o valor normalmente é definido como uma união (*union*), onde diferentes tipos de *tokens* podem ter diferentes tipos de valores (e.g. apontador para um símbolo de entrada numa tabela de símbolos).

Existem algumas variáveis pré-definidas pertencentes ao Lex e são especificadas na Tabela 3.4 baseada por [32].

Outro aspecto importante, neste processo de compilação é a caracterização do Yacc.

Como já vimos, o **Yacc (*Yet Another Compiler Compiler*)** é um programa de computador que tem o objectivo de gerar um *parser*, ou seja é a parte do compilador responsável pela análise sintática do código *source* (que contém as regras léxicas). Este *parser* é baseado especificamente pelo método LALR, na escrita analítica da gramática com uma notação similar ao BNF (*Backus-Naur Form*).

Tabela 3.4: Variáveis Pré-definidas pelo Lex.

Nome	**Função**
int yylex (void)	É chamada para invocar o *lexer*, retorna o *token*
char *yytext	Apontador para a *string* correspondente com uma expressão regular
yyleng	Tamanho da *string* correspondente com uma expressão regular
yyval	Valor associado com o *token*
int yywrap (void)	É chamada esta rotina quando o Lex alcançou o fim do ficheiro. Retorna 1 para continuar normalmente e retorna 0 para se organizar de novo.
FILE *yyout	Ficheiro de saída
FILE *yyin	Ficheiro de entrada
INITIAL	Condição inicial de arranque
BEGIN	Condição de troca de arranque
ECHO	Escreve a *string* correspondente

Um *parser* LALR é uma versão simplificada de uma analisador canónico LR[1], que separa e analisa o texto de acordo com um conjunto de regras de produção especificadas por uma gramática formal.

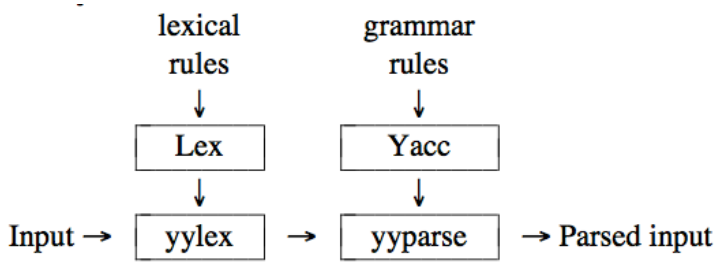
A técnica de notação BNF é uma de duas técnicas mais utilizadas para *context-free grammars* muitas vezes utilizadas para descrever a sintaxe das linguagens utilizadas em computação [30].

Uma *conext-free grammar* (CFG) é uma gramática formal em que cada regra de produção é especificada na forma "A→B", onde "A" é um símbolo não-terminal e onde "B" pode ser um símbolo terminal, não terminal ou até vazio. Um símbolo terminal ou não-terminal é assim, um elemento léxico utilizado para definir as regras que constituem a gramática formal. Não interessa que símbolos estão envolvidos, mas o único símbolo não-terminal no lado esquerdo, pode ser sempre substituído pelo respetivo do lado direito [33].

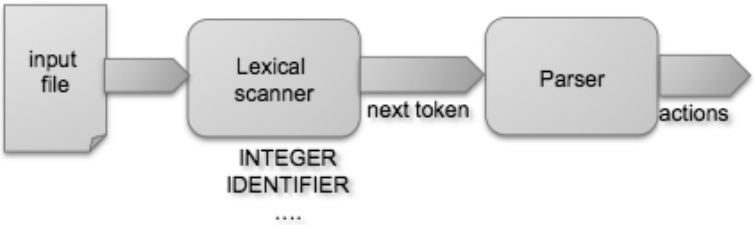
Como foi visto anteriormente, o Lex (ou *flex*) reconhece expressões regulares e o Yacc (ou *Bison*) reconhece a gramática inteira. O *Flex* divide os fluxos de entrada em peças (*tokens*), e o *Bison* pega nessas peças, e agrupa-as logicamente. Ao receber essas regras gramaticais que foram especificadas num ficheiro yacc, o *Bison* (Yacc) cria um analisador (*parser*) que reconhece "frases" válidas naquela gramática, gerando uma descrição em código C fonte, para o *parser*. O *parser* yacc fica à espera de chamar a rotina "`yylex()`" para encontrar o próximo *token*, ou seja, quando o "`yylex()`", no *Flex* (Lex), reconhecer as expressões e desempenhar as respetivas ações para cada expressão detetada, vão ser retornados os respetivos *tokens* para o *parser* criado pelo Yacc ou *Bison* (`yyparse`).

Podemos exemplificar este processo recorrendo à Figura 3.12.

[1] "LR" significa *left-to-rigth*, derivação mais à direita



(a) (retirado de [23]).



(b) (retirado de [26]).

Figura 3.12: Estrutura de funcionamento do Lex com o Yacc.

As especificações Yacc têm a mesma estrutura tripartida que as especificações Lex (pág. 41). A primeira secção é a secção das Definições, que lida com o controlo de informação para o *parser* e, geralmente, configura o ambiente de execução em que o analisador (*parser*) irá operar, ou seja, consiste nas declarações dos *tokens* e no código C necessário para incluir as bibliotecas e os ficheiros anexos à estrutura do programa (ficheiros .h, ficheiros .c, etc). A segunda secção contém a descrição das regras gramaticais com o formalismo similar à notação BNF (*Backus-Naur Form*) para a especificação da linguagem. A terceira secção contém as sub-rotinas definidas em código C.

O formalismo da gramática está dividido em três tipos de notações:

- Símbolos não-terminais, são sempre escritos em minúsculas;
  - (e.g. `expr`, `stmt`)
- Símbolos terminais (ou *tokens*), são escritos em maiúsculas ou por caracteres únicos;
  - (e.g. `INTEGER`, `FLOAT`, `IF`, `WHILE`, ';', '.')
- Regras gramaticais, produção de regras;
  - (`expr: expr '+' expr | expr '*' expr;`)
  - (E → E + E | E * E)

Os símbolos terminais (ou *tokens*) são uma classe de elementos equivalentes, do ponto de vista sintático. São representados por códigos numéricos associados aos seus identificadores, como se escree de seguida (Conteúdo adaptado de [26]):

→ Em linguagem C são correspondidos com um conjunto de `statements`

→ O *scanner* léxico "`yylex()`" deve retornar o código correspondente à classe do elemento correspondido no texto de entrada;

- A opção `-d` é utilizada para gerar o ficheiro `xx.tab.h` que contém as definições para as classes de símbolos terminais;
  - `if  return IF;` (regra léxica)
- Os símbolos terminais são declarados na secção de Definições Yacc;
  - `%token IF`
- Os *tokens* literais são utilizados para corresponder constantes de caracteres em C (e.g. '+', '*', '-'), não sendo preciso declara-los explicitamente, a menos que seja necessário especificar os tipos de dados associados, a sua precedência ou a sua propriedade associativa (*left/right*) para simplificar a gramática.
  - O código associado é a codificação ASCII correspondente;

O operador de precedência depende da ordem das declarações:
`%left '+' '-'` (menor precedência)
`%left '*' '/'` (maior precedência)
`%nonassoc UMINUS` (precedência máxima)
Considerando a especificação da regra: `expr: '-' expr %prec UMINUS`, podemos considerar que a precedência da regra é a mesma que a precedência do *token* "`UMINUS`", cuja sua precedência é a maior de todos os operadores. O "`%nonassoc`" indica que a propriedade associativa está implícita. É frequente utilizar a conjugação com "`%prec`" para especificar a precedência da regra [32].

As regras gramaticais seguem a seguinte estrutura:

```
result: components...
```

O "`result`" é o símbolo não-terminal para qual o lado direito da regra de produção é reduzida. O lado direito da regra de produção é a sequência de componentes que tanto podem consistir em símbolos terminais, como em não-terminais ou ações (código C entre {...}), por exemplo:

```
expr:  expr '+' expr  {$$=$1+$3;}
```

O símbolo `'+'` é um símbolo terminal e a ação correspondente a esta regra é "`$$=$1+$3;`". Com a recursividade à esquerda foi especificado que o programa consiste em zero ou mais expressões. Cada expressão termina com uma nova linha. Quando a nova linha é detetada é imprimido o valor da expressão. O lado direito da produção é substituído na pilha (*stack*) do *parser* pelo lado esquerdo da mesma produção. Neste caso, a regra "`expr '+' expr`" é retirada (*popping*) da pilha e a regra "`expr`" é colocada (*pushing*) na pilha.

A Figura 3.13 ilustra este processo.

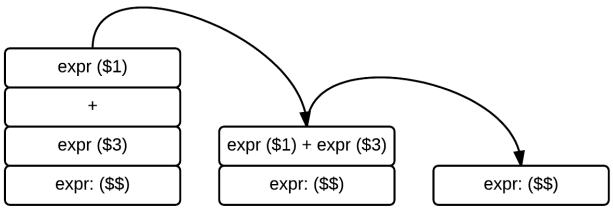


Figura 3.13: Exemplo de um processo de redução de pilha através de um "LR parser".

Este raciocínio faz com que seja reduzida a pilha, destruindo três termos e colocando um. As posições na pilha de valores na parte do código C (`$$=$1+$3;`), são referenciadas, especificando "`$1`" para o primeiro termo do lado direito da produção, "`$2`" para o segundo, por aí fora. O termo "`$$`" corresponde ao topo da pilha depois da redução ter sido feita. Sendo assim, a ação descrita, soma o valor associado a duas expressões, substitui três termos da pilha por uma soma simples, tornando o *parser* e a pilha de valores (*stack*) sincronizados.

Podem ser feitas reduções alternativas para o mesmo símbolo não-terminal, apenas separando-as com um "`|`", significando analogamente um ou "(*OR*)". Caso o lado direito da regra de produção esteja vazia, significa que a regra é satisfeita por uma *string* vazia também.

A regra é recursiva se o símbolo não-terminal do lado esquerdo ("`result`"), aparecer também no lado direito, por exemplo: `expr: expr '*' expr;`

Se diferentes tipos de dados são para ser utilizados por diferentes símbolos, a lista completa dos tipos de dados deve estar especificada nas declarações Yacc. Um dos tipos de dados declarados no código da Listagem 3.8 está associado a um símbolo terminal (`%token`) e não-terminal (`%type`).

```
%union{
        double  val;
        char    *sptr;
}
%token  <val>   NUM
%type   <sptr>  string
```

Listagem 3.8: Exemplo da utilização de um *%union* para a definição de diferentes tipos de dados (exemplo retirado de [26])

Se o valor atribuído é um apontador, então o endereço está num espaço de memória global ou numa pilha (alocação dinâmica de memória (*malloc*)).

Seguindo o exemplo da calculadora aplicado na parte do Lex na Listagem 3.7 (pág.

43), já é possível especificar as regras gramáticas e de produção correspondentes ao ficheiro de Yacc (Listagem 3.9).

```
%{
  #include <math.h>
  #include<stdio.h>
  int yylex(void);
  void yyerror(const char *);
%}
%code requires
  {
    #define YYSTYPE double
  }
%token DIGITO NUMERO
%token SOMA
%token SUBT
%token VEZES
%token DIVIDIR
%left SOMA SUBT
%left VEZES DIVIDIR
%%
S    :
     | S linha
     ;
linha   :  '\n'              {printf("\n ->Nova expressao: "); }
        |  exp '\n'          {printf("\n Valor = %f <-", $1); }
        ;
exp: NUMERO         {$$=$1;      printf("\nUsa o numero: %f", $1);}
 | exp SOMA exp     {$$=$1+$3;   printf("\nFez a Soma %f+%f", $1, $3);}
 | exp SUBT exp     {$$=$1-$3;   printf("\nFez a Subtraccao: %f-%f", $1, $3);}
 | exp VEZES exp    {$$=$1*$3;   printf("\nFez a Multiplicacao: %f*%f", $1, $3);}
 | exp DIVIDIR exp  {$$=$1/$3;   printf("\nFez a Divisao: %f/%f", $1, $3);}
 | '(' exp ')'      {$$ = $2;    printf("\nParentesis");}
 ;
%%
void yyerror(const char *s)
{
 printf("\n erro : %s",s);
}
int main(void)
{
   return yyparse();
}
```

Listagem 3.9: Ficheiro de Yacc com respetivas regras gramaticais e de produção relativo ao exemplo da calculadora.

Como se pode observar, o ficheiro Yacc correspondente às regras gramaticais da calculadora desenvolvida como exemplo, está dividido pelas três zonas principais da forma geral, onde a primeira zona das definições contém as declarações de código C necessário (código entre %{...}%), como a inclusão de bibliotecas, a definição do tipo de dados do "`yyval`", e as declarações dos *tokens* e respetivas prioridades e propriedades associativas.

Na secção das regras de produção estão definidas as regras que vão originar a árvore de derivação e a árvore sintática.

Podemos ver que um *statement* deriva numa *string* vazia ou num novo *statement* seguido de uma nova linha. A linha pode estar vazia ou conter uma expressão (`expr`). E por fim, uma expressão pode ser um número ou um resultado algébrico entre duas expressões.

As Figuras 3.14 e 3.15 apresentam a árvore de derivação e a árvore sintática, respetivamente, utilizando uma das operações entre expressões.

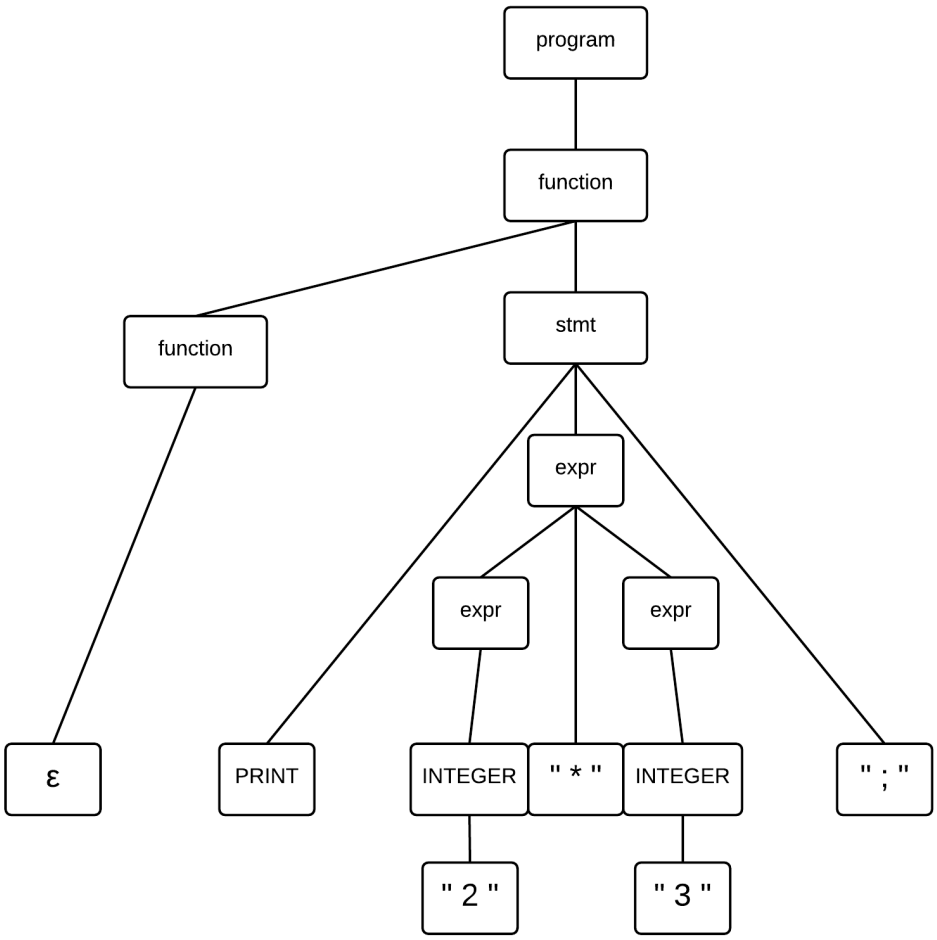


Figura 3.14: Árvore de derivação para exemplo da calculadora.

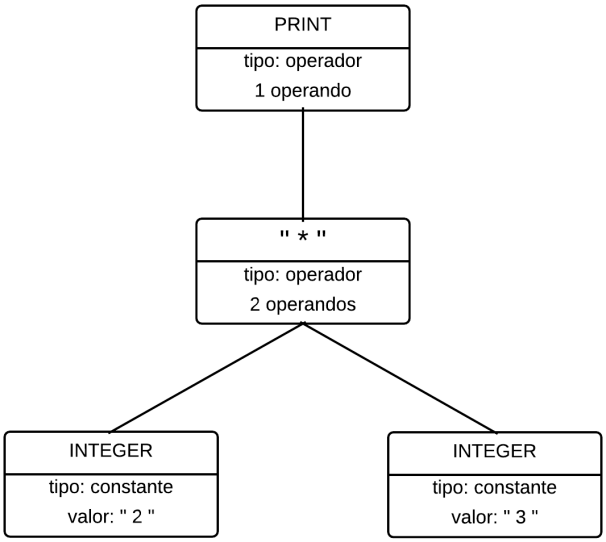


Figura 3.15: Árvore sintática para exemplo da calculadora.

Como já tinha sido referido, o Yacc utiliza as regras gramaticais, permitindo-lhe analisar os *tokens* vindos do Lex e criar a árvore de sintaxe. A árvore de sintaxe impõe uma estrutura hierárquica nos *tokens* (e.g. precedência e associatividade dos operadores).

O processo de construção de um compilador em Lex/Yacc envolve algumas etapas. A Figura 3.16 ilustra esse mesmo processo de construção de um compilador utilizando o exemplo da calculadora como modelo.

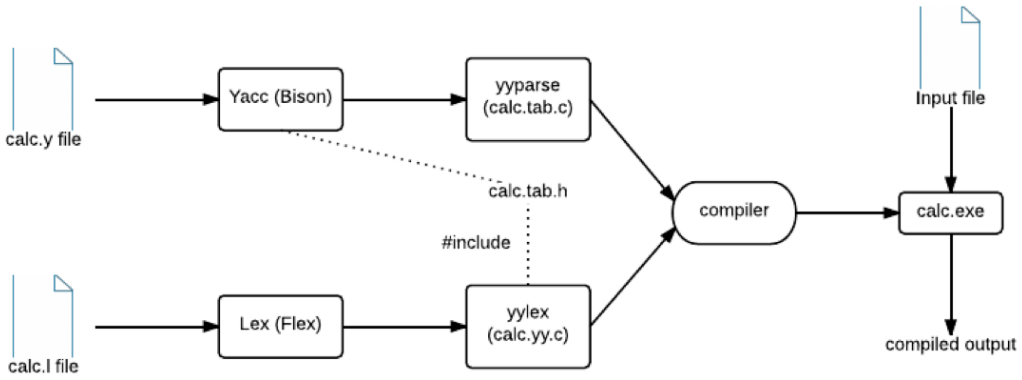


Figura 3.16: Construção de um compilador em Lex/Yacc [32].

Como se pode observar pela Figura 3.16, o Yacc vai receber e ler as descrições gramaticais definidas no ficheiro "`calc.y`", incluindo os *tokens*, e gerar um analisador sintático

(*parser*), através da função "`yyparse`" contida em "`y.tab.c`".

O Yacc vai gerar as definições para os *tokens* e colocá-las no ficheiro "`y.tab.h`". Este ficheiro vai ser lido pelo Lex juntamente com as descrições padrão do ficheiro "`calc.l`", que contém as expressões regulares, gerando um analisador léxico (*scanner/lexer*) na função "`yylex`" contido no ficheiro "`lex.yy.c`".

Por último lugar, o *lexer* e o *scanner* são compilados e interligados de forma a constituírem um executável (`calc.exe`). Para se correr o compilador, basta chamar a função "`yyparse`" que automaticamente chama a função "`yylex`" para obter cada *token*.

3.3 Implementação

3.3.1 Configuração do *Software* de Programação PL7 Junior

A implementação de código ST para o autómato TSX3721 foi realizada através do *software* de programação "PL7 Junior". Como já foi referido anteriormente, este *software* permite a programação em quatro linguagens de programação autómatos:

- Uma linguagem gráfica em *Ladder* (LD);
- Uma linguagem boleana em lista de instruções (IL);
- Uma linguagem baseada em *Grafcet*;
- A linguagem a utilizar no autómato do projeto, texto estruturado (ST).

Em todos os ambientes de programação, ou aplicações de *software* de desenvolvimento de qualquer projeto é necessário, antes de iniciar o trabalho principal, realizar um conjunto de passos para o desenvolvimento, neste caso, de uma aplicação para o autómato.

As principais etapas ou passos a seguir realizados, para o desenvolvimento da respetiva aplicação, são:

1. Criação da aplicação;
2. Definição da estrutura do programa;
3. Configuração do PLC e dos parâmetros dos módulos;
4. Estruturação de variáveis e símbolos;
5. A programação;
6. Comunicação com o PLC.

A **criação da aplicação** é iniciada com o arranque do *software PL7 Junior*, seguindo-se a seleção de um novo ficheiro através do comando "*File / New*". Faz-se a identificação do autómato programável para o qual se pretende desenvolver a aplicação, neste caso, o autómato *TSX Micro 3721 V5.0*, e como no objetivo proposto não é necessário a utilização de *Grafcet*, então seleciona-se a opção "*No*" nessa opção de *Grafcet*.

Observa-se na Figura 3.17 a selecção do PLC (*TSX7321 V5.0*).

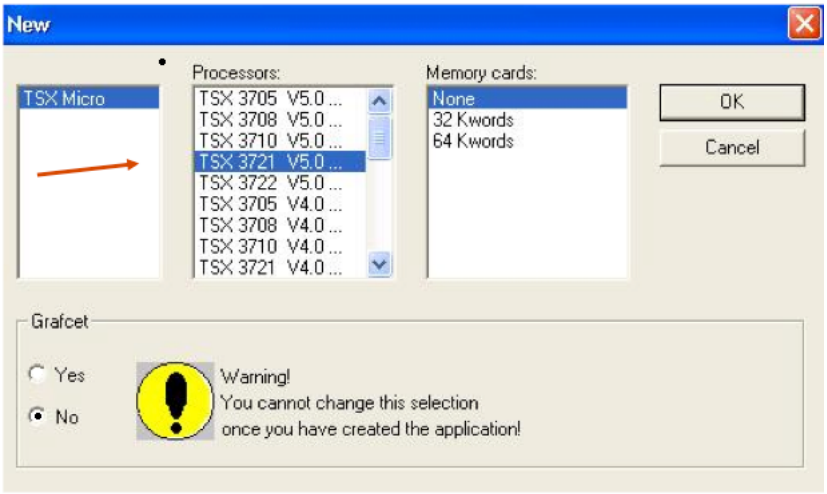


Figura 3.17: Selecção do PLC (*TSX3721 V5.0*).

A **estrutura do programa** *PL7* é constituída por secções e sub-rotinas, onde cada secção pode ser programada numa linguagem apropriada ao tipo de programa e processamento a desenvolver (ST, LD, IL). Esta divisão de secções permite criar um programa estruturado onde se pode incorporar vários módulos rapidamente. O objetivo deste projeto é a programação do autómato em texto estruturado apenas (ST), logo não é necessário incorporar outras linguagens.

A **configuração do PLC** é realizada no menu do "*Application Browser*" em "*Configuration*", depois, seleciona-se a opção "*Hardware Configuration*" para se definir os módulos instalados nas várias posições do PLC prontos a utilizar. O projeto é realizado sob entradas e saídas (I/O) discretas e portanto o módulo a escolher é o *TSX DMZ 28 DR* (como foi explicado anteriormente). Como não vai ser necessário a utilização de entradas e saídas analógicas, então, não é obrigatório selecionar o módulo *TSX AMZ 600*. Por este motivo, caso esteja instalado e não selecionado, poderá surgir uma informação de falha, mas não de avaria, e o PLC apenas funcionará com o módulo de entradas e saídas discretas.

Para a configuração dos parâmetros do programa, que por norma não se devem alterar, a opção do menu a selecionar é o *Software Configuration*.

Na Figura 3.18 está ilustrada a configuração dos parâmetros dos módulos.

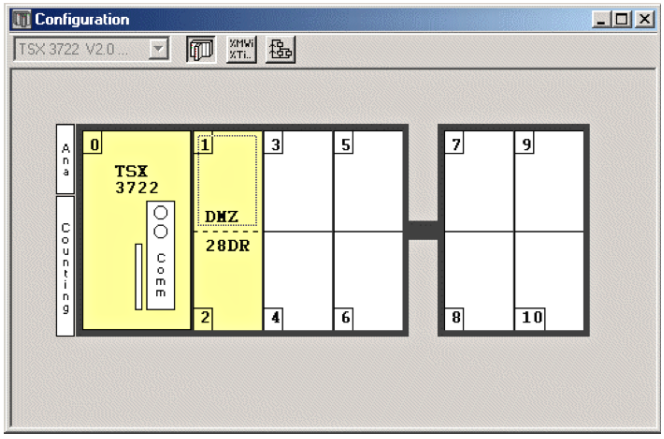


Figura 3.18: Configuração dos módulos de *hardware* do *PL7*.

A **especificação de variáveis e símbolos** é configurada e definida no editor de variáveis, que permite definir as variáveis internas e os símbolos a serem utilizados na aplicação ou programa. Ainda é possível parametrizar blocos funcionais pré-definidos (contadores, temporizadores, etc). Para se aceder ao editor de variáveis é selecionado no "*Application Browser/ STATION/ Variables/*", escolhendo depois o tipo de variável que se pretende definir (constantes, objetos do sistema ou memória, blocos funcionais pré-definidos, I/O).

A **programação** da aplicação é efectuada em Linguagem de Texto Estruturado (ST).

A **comunicação com o PLC** pode ser efetuada do PC para o PLC ou do PLC para o PC, através de um cabo de comunicação série (Terminal TER), de forma a permitir a transferência do programa. Para se conectar e trabalhar com o PLC em modo "*ON LINE*", efetua-se a ligação ao PLC (Figura 3.19) através do menu e comando "*PLC / Connect*", sendo então possível enviar os comandos de *Initialize*, *Run*, ou *Stop*.

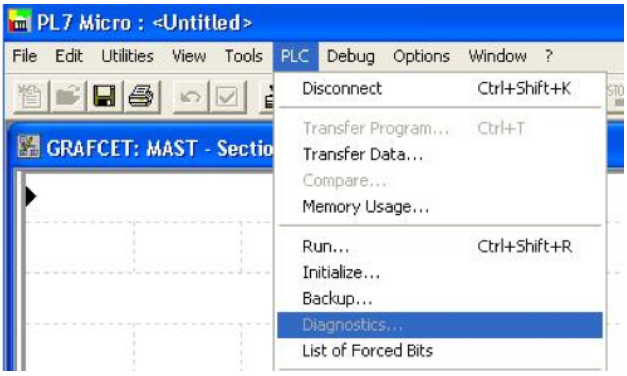


Figura 3.19: Ligação PC ↔ PLC.

3.3.2 Especificação do Código ST implementado

O Kit da máquina de lavar (SML) utilizado como modelo a seguir para testes e simulações, requer a atribuição de endereços de entrada e saída como já foi revisto anteriormente.

O código ST é a linguagem utilizada pelo autómato *TSX3721*, e como tal, uma boa programação do mesmo é fundamental para uma boa base de desenvolvimento do projeto.

Este código está dividido em quatro partes principais correspondente às quatro etapas principais do funcionamento do exemplo da máquina de lavar:

1. A ação "IN" (entrada de água) enquanto o nível "H2" não é atingido;
2. Acção de "N2" (rotação do motor "M") durante 10 segundos;
3. Acção "OUT" (esvaziamento da cuba) durante 15 segundos;
4. Paragem de emergência.

A variável relacionada com a entrada da água no tanque é a variável `%Q2.3`. O botão de impulso de marcha é o `%I1.6`, o sensor do nível "H2" é o `%I1.3`, como tal, o código que representa a primeira etapa da máquina de lavar, está ilustrado na Listagem 3.10.

A variável correspondente à ação "IN" (`%Q2.3`) ficará com valor lógico "1" quando o botão de impulso de marcha `%I1.6` for acionado (`%I1.6` = "1"). De modo a se garantir que a entrada de água não ocorre quando se acciona o botão de paragem de emergência e quando se atinge o nível "H2" é adicionado a condição de "`AND NOT`" com os respetivos endereços do botão de paragem (`%I1.5`) e sensor de nível "H2" (`%I1.3`). Como os requisitos da máquina de lavar propõem botões de impulso é necessário programar esse comportamento, bastando apenas utilizar recursividade na atribuição da variável de saída, tornando-a ativa quando esta mesmo ficar activa uma primeira vez.

```
%Q2.3:=(%I1.6 OR %Q2.3)AND NOT %I1.5 AND NOT %I1.3;
(*Action IN - Until H2*)
```

Listagem 3.10: Acção de entrada de água no tanque.

As duas seguintes etapas do processo baseiam-se no funcionamento de um temporizador "TP", segundo o qual, a ação de funcionamento é executada durante um intervalo de tempo.

A segunda etapa (Listagem 3.11) corresponde à rotação de um motor "M" por ação de "N2" (`%Q2.0`) durante 10 segundos. Neste caso, o temporizador é iniciado (`START %TM1`) quando o sensor de nível "H2" (`%I1.3`) é acionado. Após os 10 segundos, aproximadamente, o temporizador é desativado (`DOWN %TM1`). Durante este intervalo de tempo, de

modo a manter a rotação do motor, o endereço correspondente a "N2" (`%Q2.0`) tem de estar atribuído à saída do temporizador (`%Q2.0 := %TM1.Q`).

```
IF %I1.3 THEN
        START %TM1;
ELSIF(%TM1.V>=10)THEN
        DOWN %TM1;
END_IF;
%Q2.0:=%TM1.Q;
(*MOTOR M, N2*)
```

Listagem 3.11: Funcionamento do motor M (N2) durante um intervalo de tempo de 10 s.

A ação de esvaziamento da cuba ("OUT" - `%Q2.4`), durante 15 segundos é também ela programada baseada num temporizador "TP"e está apresentada na Listagem 3.12.

À semelhança da fase anterior, o esvaziamento ocorre durante um certo período de tempo, neste caso 15 segundos, e portanto, o comportamento é muito semelhante. Existe um temporizador (TM3) que é iniciado quando o temporizador (TM1) anterior terminar (`%TM1.V >= 10`), e um *Reset* que bloqueia o funcionamento do motor "M" e correspondente ação "N2" (`RESET %Q2.0`) no mesmo instante. O temporizador é desativado após os 15 segundos de funcionamento como inicialmente proposto (`DOWN %TM3`). A ação de esvaziamento está decorrendo enquanto os 15 segundos não pararem, e portanto, o endereço de saída tem de estar relacionada com o endereço do temporizador (TM3 - `%Q2.4 := %TM3.Q`).

```
IF %TM1.V>=10 THEN
        START %TM3;
        RESET %Q2.0;
ELSIF(%TM3.V>=15)THEN
        DOWN %TM3;
END_IF;

%Q2.4:=%TM3.Q;
(*ACTION OUT*)
```

Listagem 3.12: Temporizador para ação do esvaziamento da cuba.

Como segurança, ou para simular um modo de segurança é implementado um mecanismo que permite interromper o funcionamento do sistema, deste modo, sempre que o botão de impulso, referenciado como "`%I1.5`", for acionado, os temporizadores devem parar, e como tal, todos os mecanismos dependentes dos mesmos, devem parar também. A Listagem 3.13 apresenta este mecanismo em código ST.

```
%Q2.6:=%I1.5 OR %Q2.6;
IF %Q2.6 THEN
        DOWN %TM1;
        DOWN %TM3;
        RESET %Q2.3;
END_IF;
```

Listagem 3.13: Mecanismo de paragem de emergência na implementação do código ST.

É de notar que o botão de impulso é simulado através de um mecanismo de recursividade de endereços, neste caso o endereço auxiliar "`%Q2.6`" (`%Q2.6 := %I1.5 OR %Q2.6`), permitindo criar esse mesmo efeito de impulso, uma vez que o Kit testado no laboratório, tem apenas comutadores de 2 posições, não cumprindo inicialmente com os requisitos propostos. Se esse comutador for acionado, então os temporizadores TM1 ou TM3 serão desativados, e a respetiva ação de entrada de água (IN) é também interrompida, restabelecendo o correspondente endereço `%Q2.3`.

Os testes do código no kit do autómato utilizado no projeto, estão especificados e explicados no capítulo "Resultados Experimentais".

3.3.3 Implementação do Compilador Baseado em Lex / Yacc

Um compilador baseado nas tecnologias de Lex e Yacc, como foi revisto anteriormente, necessita de duas fontes de código, o *Source Lex* e o *Source Yacc*, uma vez que o Lex é utilizado juntamente, neste caso, com um interpretador *parser* para executar a fase de análise léxica.

O Lex é gerador de programa para processamento léxico de fluxos de entrada de caracteres.

O código *Source Lex* utilizado no projeto é denominado de "`thesis.l`" e representa as ações e expressões especificadas pelo utilizador, ou seja, as expressões regulares especificadas dadas depois ao Lex.

O formato utilizado no *Lex Source* do projeto ("`thesis.l`") apresenta o formato geral deste tipo de códigos, ou seja:

%{
Declarações em C
%}
Definições
%%
{*Regras*
%%
Sub-rotinas

Apesar de muitas vezes as "Definições" ou as "sub-rotinas" estarem omitidas, no caso concreto do projeto, foi necessário definir algumas bibliotecas e incluir alguns ficheiros com estruturas importantes para a compilação e obtenção dos resultados. As declarações em C utilizadas são mostradas na Listagem 3.14.

```
%{
  #include <stdlib.h>
  #include <string.h>
  #include "type_struct.h"
  #include "thesis.tab.h"
  void yyerror(char *);
%}
```

Listagem 3.14: As "Definições" declaradas no *Lex Source*.

O formato da estrutura apresentada apresenta separadores (%%) entre "Definições", "Regras" e "Sub-rotinas". Após as "Definições" vem a secção mais importante do ficheiro léxico, as "Regras". As Regras representam decisões de controlo do utilizador, estão estruturadas como uma tabela, em que a coluna da esquerda contém expressões regulares e a coluna direita contém as respetivas ações, partes do programa a ser executado quando as expressões são reconhecidas.

No código que se segue (Listagem 3.15) é possível ver alguns exemplos de regras implementadas no projeto, com o objetivo de reconhecer algumas expressões características da linguagem em Texto Estruturado (ST).

```
%%
0                { yylval.iValue = atoi(yytext);  return INTEGER; }
[1-9][0-9]*      { yylval.iValue = atoi(yytext);  return INTEGER; }
[-()<>=+*/;{}.] { return *yytext; }
":="             return AT;
">="\space               return GE;
"=="\space               return EQ;
"!="             return NE;
"while"          return WHILE;
"if"             return IF;
"then"           return THEN;
"elsif"          return ELSE;
"end_if"         return ENDIF;
"or"             return OR;
"and"            return AND;
"START"          return START;
"DOWN"           return DOWN;
"TOF"            { yylval.iValue = PL7_TOF;       return INTEGER;}
"TP"             { yylval.iValue = PL7_TP;        return INTEGER;}
\%([Ii]){1}      {strcpy(yylval.strval, yytext); return DIGITAL_INPUT;}
\%([Qq]){1}      {strcpy(yylval.strval, yytext); return DIGITAL_OUTPUT;}
%%
```

Listagem 3.15: Exemplos de regras utilizadas na implementação do *Lex Source*.

Como é necessário utilizar o Lex e o Yacc, e o Yacc chama a rotina *yylex* definida no Lex. Podemos observar algumas regras do código apresentado (Listagem 3.15) onde são feitas as correspondências de reconhecimento de algumas instruções da linguagem ST (Ex: `"If" return IF;`). O analisador léxico retorna *tokens* como `INTEGER`, `IF`, `ELSE` ou `DIGITAL_INPUT`, que ficam representados como símbolos ou identificadores com um respetivo número associado.

O ficheiro criado no projeto para representar as descrições e especificações gramaticais é o "`thesis.y`". A forma da estrutura é análoga à do ficheiro "`thesis.l`", que contém todas as especificações para o padrão de correspondência de regras para o Lex. Sendo assim, vamos considerar que a estrutura também se baseia em "Declarações C e Definições Yacc", "Regras" e "Sub-rotinas", e todas elas separadas pelos separadores "%%". As declarações em C são muito parecidas às do ficheiro de Lex ("`thesis.l`"), contendo apenas mais algumas bibliotecas necessárias. Ainda nas declarações é de notar a existência de duas inicializações de variáveis e a declaração de vetores ou matrizes, pertencentes a funções ou estruturas utilizadas mais adiante.

Como complemento necessário e decisivo à organização e estrutura deste projeto de compilação foi criado um ficheiro de cabeçalho ("`type_struct.h`") que representa um segundo ficheiro fonte, onde estão definidas as principais estruturas, protótipos de funções e definição de variáveis, neste caso, dos tipos de instruções diferentes, utilizados ao longo das especificações da linguagem ST do projeto. O *Header file* ("`type_struct.h`") especifica uma estrutura principal chamada "`nodeType`", no qual, todos os seus elementos podem também corresponder a uma outra estrutura. A caracterização da estrutura "`nodeType`" é apresentada na tabela 3.5.

Tabela 3.5: Estruturas dos tipos de símbolos e funções utilizadas nas produções de regras gramaticais.

nodeType	**idNoteType**	**conNodeType**	**oprNodeType**	**nodeEnum**
(nodeEnum) Type	(int) i	(int) value	(int) oper	typeCon
(id) last_edge	(int) channel	-	(int) nops	typeId
(conNodeType) con	(int) bit	-	(nodeType) **op	typeOpr
(idNodeType) id	-	-	-	**PL7***
(oprNodeType) opr	-	-	-	-

O elemento "**PL7***" assinalado, pertencente à estrutura "`nodeEnum`", corresponde a um conjunto de elementos destinados a representar entradas, saídas e configurações de temporizadores, para linguagem ST, como o *software PL7* o faz. Esta estrutura ("`nodeEnum`") enumera um conjunto de vários tipos de símbolos utilizados nas regras e ações gramaticais definidas no ficheiro de yacc.

As regras gramaticais podem estruturar-se em três grupos principais:

- Grupo Identificadores, para corresponder variáveis, constantes, a uma determinada posição de um vetor (estrutura "`idNodeType`");

- Grupo Constantes, que contém o valor da respectiva constante, variável (estrutura "`conNodeType`");

- Grupo Operador, constituído pelo tipo de operador (`oper`), pelo número de operandos (`nops`) e pelos próprios operandos (`**op`), estes últimos que podem ser recursivamente outra estrutura "`nodeType`" novamente.

Sendo assim, a estrutura "nodeType" é constituída por elementos que podem ser também eles outras estruturas.

Como vimos, o *Header file* ("`type_struct.h`") contém a definição destas estruturas apresentadas e todos os protótipos de funções utilizadas e chamadas nas ações gramaticais. Estas funções estão construídas e configuradas num outro ficheiro auxiliar, "`nodes_routines.c`", que implementa as funções correspondentes ao preenchimento de todas as definições das estruturas anteriormente apresentadas, ou seja, define as funções para definir uma nova constante, um novo operador, um novo identificador, um identificador de um temporizador, uma função para obter valores, etc, configura as rotinas utilizadas nas definições gramaticais do ficheiro de yacc.

Os protótipos das funções agora mencionadas, podem ser verificados na Listagem 3.16.

```
/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(char *var_nome);
nodeType *id_input(int channel, int bit);
nodeType *id_output(int channel, int bit);
nodeType *id_timer_conf(int channel, char *str);
nodeType *id_timer(int channel);
nodeType *id_timer_config(int timer_number, char *timer_parameter);
nodeType *con(int value);

void    freeNode(nodeType *p);
int     ex(nodeType *p);
int     yylex(void);
void    yyerror(char *s);
long    CurrenTimeMili();

int     set_value(char *variable, int value);
int     get_value(char *variable);

void    set_timer_parameter(int time_number, char *parameter, int value);
int     get_timer_parameter(int time_number, char *parameter);
int     timer_behavior(int time_number, int parameter);
```

Listagem 3.16: Protótipos das funções implementadas no ficheiro (type_structed.h).

Agora que foram apresentadas os protótipos das funções e as estruturas dos elementos a utilizar na gramática, podemos voltar a atenção para o ficheiro yacc, e as suas implementações. Como foi visto, a grande diferença para este ficheiro, recai nas regras e nas ações implementadas, contendo todas as descrições gramaticais para a especificação da linguagem a utilizar, neste caso a linguagem ST. A secção das regras gramaticais e ações associadas está dividida em três partes principais:

- Símbolos terminais;
- Símbolos não terminais;
- Regras gramaticais (regras de produção).

Os símbolos terminais e não terminais são declarados na secção de declarações do Yacc. São representados por códigos numéricos associados ao seu identificador. O código correspondente à classe do elemento selecionado no texto de entrada foi retornado pelo analisador léxico (*yylex()*). Os símbolos terminais, não terminais e respetivas propriedades declaradas no projeto podem ser consultados na Listagem 3.17.

```
%union {
   int iValue; /* integer value */
   char strval[256];
   nodeType *nPtr; /* node pointer */
};

/* Terminal Symbols */c
%token <iValue> INTEGER
%token <strval> STR
%token DIGITAL_INPUT DIGITAL_OUTPUT
%token TIMER
%token <strval> TIMER_PARAMETER
%token WHILE IF PRINT SQUARE START DOWN THEN ENDIF OR RESET AND NOT AT

/* Non-terminals Symbols */
%type <nPtr> stmt expr stmt_list identificador canais timers timer_parameter
    timer_id

/* Properties */
%nonassoc IFX
%nonassoc ELSE THEN ENDIF
%left GE LE EQ NE '>' '<' OR AND NOT
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS RE FE
```

Listagem 3.17: Definições Yacc (símbolos terminais, não terminais e propriedades).

Os diferentes tipos de dados utilizados para diferentes símbolos podem ser acoplados numa lista (*union*), ocupando cada um deles o mesmo *slot* de memória, não podendo ser utilizados em simultâneo. O símbolo terminal, como o caso do *token* "<ivalue>

INTEGER", corresponde a um valor inteiro como tipo de dado, e o símbolo não terminal, como o caso do `%type <nPtr> stmt`, corresponde a uma declaração de uma possível instrução, sendo portanto, um apontador para uma estrutura do tipo "nodeType". Os *literal tokens* são utilizados para corresponder caracteres diretamente escritos em C (e.g. `'+'`, `'-'`), e como se pode ver na Listagem 3.17 não é necessário declará-los explicitamente, a não ser a necessidade de especificar a sua precedência ou o tipo de propriedade associativa (*left / right*) para os operadores, simplificando a gramática.

A implementação das regras gramaticais segue a estrutura formal típica numa estrutura yacc. A estrutura das regras de produção e gramaticais corresponde a uma sequência de componentes com símbolos terminais, não terminais e ações (funções em código C).

Na Listagem 3.18 são apresentadas as primeiras definições e respectivas reduções das regras de produção correspondentes, onde o programa é constituído por funções que executam instruções. Uma função pode ser constituída por nada ou por outras funções e instruções (*statements* - (stmt)). Cada stmt especifica as instruções mais utilizadas em linguagem ST, e são apresentados alguns exemplos utilizados na implementação do projeto. Entre muitos outros exemplos e instruções implementadas podemos observar que cada *stmt* pode ser uma expressão, um conjunto de condicionais ou a atribuição de valores lógicos a variáveis por exemplo.

```
%%
program:
  function {
   void execute_statements(void);
   execute_statements();
  } ;
function:
   function stmt {
    statements[statements_count]=$2;
    statements_count++;
   }
   |  /* NULL */        {statements_count=0;} ;
stmt:
  ';'                                   {$$=opr(';', 2, NULL, NULL);}
   | expr ';'                           {$$=$1;}
   | START timer_id  ';'                {$$ = opr(START, 1, $2);}
   | identificador ASSIGN expr ';'      {$$=opr('=', 2, $1, $3);}
   | '{' stmt_list '}'                  {$$=$2;}
   | if_stmt                            { $$ = $1;}
```

Listagem 3.18: Implementação prática das definição das regras gramaticais e respetivas ações em Yacc (parte 1).

O próximo conjunto de regras gramaticais e de produção dão seguimento à estrutura da definição de regras utilizada. Apesar de cada "stmt" ser um dos símbolos não terminais mais importante das definições, também se pode ramificar em outras definições segundo o mesmo padrão BNF (*Backus-naur Form*). Assim sendo, mais alguns exemplos são

ilustrados (Listagem 3.19) relativamente à implementação realizada. É possível verificar como pode ser dividida uma expressão (`expr`), um identificador (`identificador`), os canais, ou até temporizadores. Cada um destes símbolos, pode por sua vez, representar novos encadeamentos. Pelas regras básicas da programação, em particular pela linguagem ST, cada instrução ou conjunto de instruções (*statement*), pode corresponder recursivamente a outro conjunto de instruções ou uma lista com vários conjuntos de instruções (`stmt`, `stmt_list stmt`). Como as instruções condicionais (`IF, ELSE, ELSIF`) têm comportamentos parecidos mas ligeiramente diferentes, foi necessário dividir as suas definições e regras em algumas ramificações ou símbolos não terminais (`if_stmt`, `else_part`). Vimos que um *statement* pode ser uma expressão apenas (`expr`), então, cada expressão pode corresponder a um número inteiro (`INTEGER`), a um identificador (variáveis de entrada e saída (`canais`), *timers*, apenas uma *string*), ou a um conjunto de instruções e operações aritméticas e lógicas (e.g. `expr OR expr`). Os temporizadores (*timers*) são definidos e divididos pelos seus parâmetros (`timer_parameter`) e pela sua identificação (`timer_id`). A Listagem 3.19 apresenta alguma da implementação prática referente às regras gramaticais e respetivas ações em Yacc.

```
if_stmt:
   IF expr THEN stmt_list else_part END_IF ';'   {$$=opr(ELSIF, 3, $2, $4, $5);}
        ;
else_part:                                       {$$ = NULL; /*epsilon*/}
   | ELSE  stmt_list                             {$$ = opr(ELSE, 1, $2);}
   | ELSIF expr THEN stmt_list else_part         {$$=opr(ELSIF, 3, $2, $4, $5);}
       ;
stmt_list:
        stmt                                     {$$=$1;}
        | stmt_list stmt                         {$$=opr(';', 2, $1, $2);} ;
expr:
        INTEGER                                  {$$=con($1);}
        | identificador                          {$$=$1;}
        | expr OR expr                           {$$=opr(OR, 2, $1, $3);};
identificador:
        STR                                      {$$=id($1);}
        | canais                                 {$$=$1;}
        | timers                                 {$$=$1;} ;
canais:
   DIGITAL_INPUT  INTEGER '.' INTEGER            {$$=id_input($2,$4);}
   | DIGITAL_OUTPUT INTEGER '.' INTEGER          {$$=id_output($2,$4);} ;
timers:
        timer_parameter                          {$$=$1;}
        | timer_id                               {$$= $1;} ;
timer_parameter:
   TIMER INTEGER '.' TIMER_PARAMETER             {$$=id_timer_config($2,$4);} ;
timer_id:
   TIMER INTEGER                                 { $$ = id_timer($2);} ;
%%
```

Listagem 3.19: Implementação prática da definição das regras gramaticais e respetivas ações em Yacc (parte 2).

Agora que estão definidas as principais regras gramaticais que definem a sintaxe da linguagem em Texto Estruturado é necessário especificar as ações correspondentes a cada regra de produção, para se definir a semântica do código lido.

A ação é um bloco em código C que é executado quando a regra de produção é aplicada (redução), e normalmente aparece entre os símbolos na sequência do lado direito da regra de produção. Na implementação do projeto algumas das ações das regras de produção foram programadas como funções em código C. É de notar nas Listagens 3.18 e 3.19, que a sequência de código do lado direito das regras de produção (e.g. `{$$ = opr(OR, 2, $1, $3);}`) é quase sempre definida por uma função em C. Neste exemplo a função retrata uma operação lógica com o elemento "OR", o segundo parâmetro corresponde ao número de operandos e os dois últimos campos, aos próprios operandos envolvidos (ver cabeçalho da função na Listagem 3.16, especificação pode ser lida na mesma página no tópico "Operador").

O termo "`$1`" representa a primeira posição do lado direito da produção e "`$3`" representa o segundo termo, estando este na 3ª posição. No meio dos dois termos, está neste caso, o *token* representativo da instrução lógica "`OR`".

Na Figura 3.20 é ilustrada a árvore de derivação (*Derivation Tree*) referente a uma definição da regra de produção do símbolo não terminal "`if_stmt`". Este símbolo representa a caracterização e a estrutura gramatical de uma instrução condicional na linguagem ST ("`IF`", "`ELSE`", "`ElSIF`", "`END_IF`").

As regras de produção implementadas para este tipo de instruções na Listagem 3.20, especificam a árvore de derivação correspondente e ilustrada na Figura 3.20.

```
program:
  function {
   void execute_statements(void);
   execute_statements();
  } ;
function:
   function stmt {
    statements[statements_count]=$2;
    statements_count++;
   }
   |  /* NULL */        {statements_count=0;} ;
stmt:
   | if_stmt                              { $$ = $1;}
if_stmt:
    IF expr THEN stmt_list else_part END_IF ';' {$$=opr(ELSIF, 3, $2, $4, $5);}
         ;
else_part:                                      {$$ = NULL; /*epsilon*/}
    | ELSE  stmt                                {$$ = opr(ELSE, 1, $2);}
    | ELSIF expr THEN stmt_list else_part       {$$=opr(ELSIF, 3, $2, $4, $5)
       ;};
```

Listagem 3.20: Definição das regras de produção de um *IF Satement*.

Analogamente e considerando o mesmo exemplo de aplicação podemos definir a árvore sintática construída internamente pelo *parser* (Figura 3.21).

A árvore sintática é gerada pelo *parser* e representa a estrutura gramatical de um bloco ou conjunto de instruções a realizar no momento. Enquanto que a árvore de derivação (Figura 3.20) ilustra em como é que são derivados os vários símbolos não terminais, a árvore sintática (Figura 3.21) apresenta os argumentos constituintes de cada instrução (neste caso de um *IF statement*).

Na ação de produção, como já foi referenciado, existe sempre associado uma função programada em C. Olhando para as árvores de derivação e sintática criadas internamente e ilustradas nas Figuras 3.20 e 3.21, respetivamente, podemos reparar que no topo dessas árvores, encontra-se a ação de produção a executar. Essa ação, que está especificada no símbolo não terminal "`function`" é uma função também ela programada em C definida num ficheiro auxiliar responsável pela interpretação das ações de produção definidas e prontas a executar.

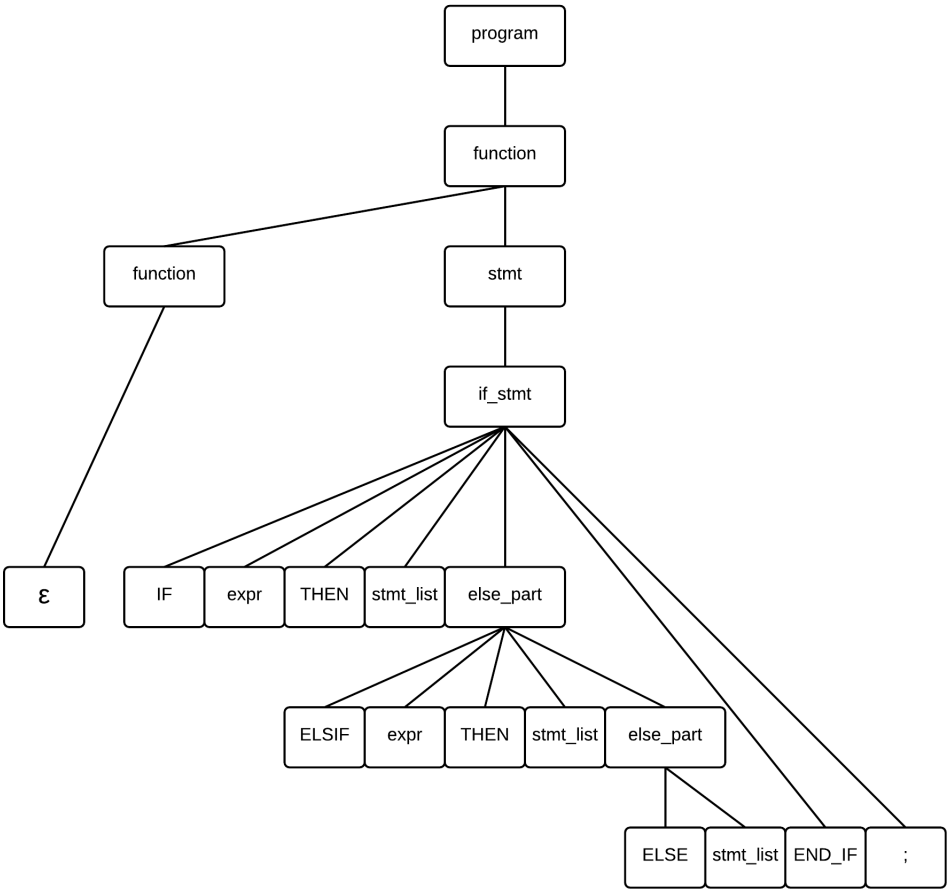


Figura 3.20: Árvore de derivação de uma instrução "IF".

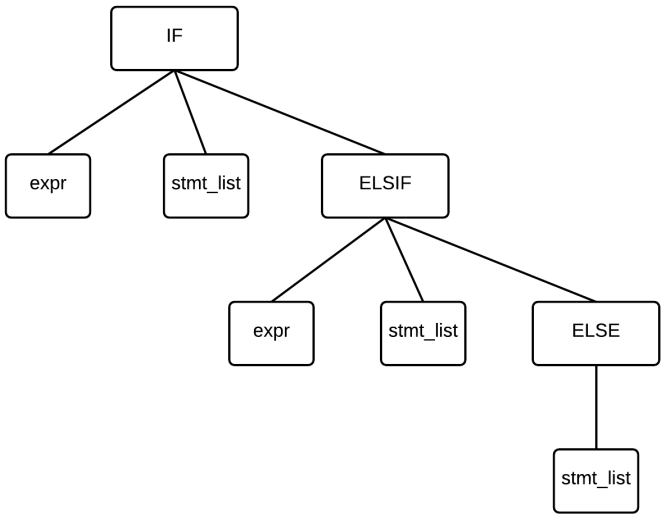


Figura 3.21: Árvore sintática de uma instrução "IF".

Este ficheiro auxiliar é denominado por "`interpreter.c`". Como interpretador contém o raciocínio lógico de cada tipo de função a executar, consoante os vários tipos de dados definidos nas respetivas estruturas anteriormente apresentadas (Tabela 3.1). O símbolo "`function`" é definido pela função "`execute_statements();`" cuja função é definida no interpretador "`interpreter.c`", e tem o objetivo de percorrer todos os *statements* definidos ao longo do programa e executá-los (`ex(statements[i],"i"` é um dos *statements* do programa a percorrer, ou seja, uma estrutura do tipo "*nodeType*").

Assim sendo, pegando no exemplo de ação de produção definida pela função "`$$ = opr(IF, 3, $2, $4, $5);`", a função "`ex();`" recebe-a como argumento, porque `opr(...)` é considerado um *statement (stmt)* (ver árvore de derivação em Figura 3.20) e analisa que tipo de estrutura está a lidar. Neste caso o tipo de estrutura é "`typeOpr`", uma vez que estamos a lidar com uma operação, e portanto vai verificar o tipo de operador (neste exemplo é o "IF"). Após verificado o operador, o interpretador vai executar em C as instruções desejadas e previamente programadas.

O código apresentado na Listagem 3.21 elucida este mesmo raciocínio anteriormente especificado. Podemos verificar pelo código (Listagem 3.21) mostrado, que no caso do tipo de estrutura ser um operador (`p→type` e `typeOpr`) e o tipo de operador ser um "`IF`" ou um "`ELSIF`" como é o caso, que a função "`ex();`" vai sendo chamada recursivamente consoante as posições dos operandos, e depois o seu tipo de estrutura respetivamente, utilizando em código C as regras das instruções condicionais `if` e `else`.

Não esquecer que a expressão que define a condição do "`IF`", ou seja "`IF expr...`", será à partida,uma variável de entrada (*input*) ou de saída (*output*), sendo um tipo de estrutura pertencente a um identificador (`typeId`). Pela mesma lógica serão realizadas e

```
void execute_statements() {
        for (int i = 0; i < statements_count; i++) {
                ex(statements[i]);
        }
}
int ex(nodeType *p) {
        if (!p) return 0;
        switch(p->type) {
                case typeOpr:
                        case IF:
                        case ELSIF:
                                if (ex(p->opr.op[0]))
                                        ex(p->opr.op[1]);
                                        else ex(p->opr.op[2]);
                                return 0;
                        case ELSE:
                                ex(p->opr.op[0]);
                                return 0;
                return 0;
        return 0;
}
```

Listagem 3.21: Tomada de decisão e respetiva execução por parte do interpretador (*interpreter.c*).

executadas as respetivas instruções associadas ao "typeId". Em relação ao *statement* associado à instrução a realizar após o "THEN" poderá, e será com certeza, uma nova operação que poderá ter, por sua vez, instruções com identificadores (typeId), com constantes (typeCon), ou até novas operações incluindo novas instruções condicionais (IF's, etc).

O tipo de estrutura "typeOpr" envolve um conjunto de operadores muito elevado. São definidas operações aritméticas, lógicas, com temporizadores, com *Resets*, entre outros. Para se perceber melhor os tipos funções, instruções e operações a executar, a Tabela 3.6 especifica resumidamente os tipos de operações a interpretar e executar, respetivamente.

É no interpretador ("interpreter.c") que estão definidas as funções a exportar pela DLL a ser utilizada na interface gráfica de utilizador programada em C# recorrendo ao *software Microsoft Visual Studio*. Para se poder exportar as funções através da DLL usou-se a palavra chave "_declspec (dllexport)", que adiciona a diretoria de exportação para o arquivo objeto. As funções definidas para a DLL no interpretador estão especificadas na Tabela 3.7.

Por fim, e tendo toda a parte do Lex e Yacc definida e configurada é possível compilar todos os ficheiros de modo a se conseguir obter um reconhecedor capaz de testar e interpretar um código escrito em texto estruturado (ST). Nas sub-rotinas do ficheiro yacc ("thesis.y"), a função "main"chama outra função "fazer_o_trabalho()", que configura e define o tipo de leitura de ficheiro do código ST, se é em modo leitura do teclado (yyin) ou se é a partir de algum ficheiro exterior com o código ST. Contém também a

Tabela 3.6: Tipos de operações a executar pelo interpretador.

typeOpr	**Funcionalidade**
PL7_RE	Representa um *Rising Edge* de um temporizador, identificando a mudança de estado de 0→1.
PL7_FE	Representa um *Falling Edge* de um temporizador, identificando a mudança de estado de 1→0.
GE/LE/NE/EQ	Correspondem a operações de comparação entre duas expressões, como ">=", "<=", "! =" ou "==".
START	Inicia o comportamento do temporizador definindo anteriormente, configura o seu número de identificação, o seu tipo de parâmetro ("V", "Q", "P", etc) e o próprio valor de cada parâmetro. A função responsável pelas definições do *timer* é o **set_timer_ parameter()** e o **get_timer_parameter()**, e a função responsável por todo o comportamento do temporizador é o **timer_behavior()**.
DOWN	É igual à operação START, mas o valor de ordem do temporizador é igual a 0 em vez de 1.
'S'	Força o valor da variável sempre a 1.
'R'	Força o valor da variável sempre a 0.

Tabela 3.7: Funções exportadas para a DLL.

_declspec(dllexport)	**Funcionalidade**
run_st_code()	Percorre todos os *statements* e dá ordem de decisão de execução pela função **ex()**. Retorna o número de *statements*.
free_st_code()	Liberta as estruturas dos *nodes* criados ao longo do programa, reinicia o número de *statements* e o tamanho do vector **sym** que contém o índice das variáveis e o seu respectivo tamanho.
parse_structured_text()	Similar a uma função *main*, contém algumas funções típicas do yacc.
set_sym_value()	Exporta a função **set_value** que devolve o índice da variável.
get_sym_value()	Exporta a função **get_value** que devolve o valor da variável.

função "`yyparse()`"que devolve 1 caso ocorra algum erro nas definições da gramática. Esta função automaticamente chama a função "`yylex()`"para obter os *tokens*.

Para se conseguir obter um executável, capaz de representar o compilador de texto estruturado, foi necessário executar alguns comandos que traduzem o processo de construção de um compilador Lex/Yacc. Os comandos implementados e ilustrados na Listagem 3.22 elucidam os passos do processo de construção do compilador (rever Figura 3.16).

Resumindo, neste processo de compilação e funcionamento geral do Lex e Yacc no projeto, o Yacc lê as gramáticas descritas pelo "`thesis.y`" e gera um analisador sintático (*parser*) que inclui a função "`yyparse()`"no ficheiro "`thesis.tab.c`". O ficheiro "`thesis.y`", contém as declarações dos *tokens*.

A opção "`-d`" faz com que o yacc gere as definições para os *tokens* e os coloque no ficheiro "`thesis.tab.h`".

O Lex lê as descrições padrão realizadas no "`thesis.l`", e incluindo o ficheiro "`thesis.tab.h`", gera um analisador léxico (*lexer*) que inclui a função "`yylex()`"no ficheiro "`lex.yy.c`".

Por fim, o analisador léxico e sintático são compilados e interligados, para se criar o executável "`thesis.exe`".

O código representativo deste processo é elucidado na Listagem 3.22.

```
rem -i no flex to insensitive case

# create y.tab.h, y.tab.c
flex -i thesis.l

#create lex.yy.c
bison -d thesis.y

#compile/link
cl lex.yy.c thesis.tab.c node_routines.c interpreter.c -o these.exe
```

Listagem 3.22: Construção do compilador Lex / Yacc.

3.3.4 Interface Gráfica de Utilizador (*GUI*) em C#

O sistema de compilação referente ao interpretador do código de linguagem em texto estruturado e a interface gráfica de utilizador, necessária à realização de um simulador do mesmo, são utilizados e desenvolvidos maioritariamente através da plataforma de programação *Visual Studio (2013)*.

A escolha deste *software* recaiu, principalmente, pela fácil e prática criação de interfaces gráficas gráficas de utilizador (*GUI - Graphical User Interface*).

No mundo da computação, uma "*GUI*", é um tipo de interface que permite utilizadores interagirem com dispositivos eletrónicos através de *icons* e vários indicadores visuais, ao contrário das interfaces baseadas em texto [7].

O *Visual Studio* é um ambiente de desenvolvimento integrado (IDE) da *Microsoft*, utilizado para desenvolver programas de computador em ambiente *Windows*, como *web sites*, aplicações *web* e plataformas de desenvolvimento de *software* como o *Windows Form*. O *Windows Form Designer* é utilizado para construir aplicações *GUI*, cuja disposição da estrutura (*layout*) pode ser controlada, colocando os controlos dentro de outros *containers*[2], ou bloqueando-os para o lado da *form*[3] [28].

[2]Em programação orientada a objectos, um *container* é um objecto que contém um ou mais objectos

[3]Em computação, *form* é um termo utilizado para descrever a interface de um aplicativo, normalmente entre cliente e servidor

A interface do utilizador (*UI*) da *GUI* está relacionada com o código em si, utilizando um modelo de programação orientada a objectos. O *designer* vai gerar código C# para a aplicação.

De modo a se conseguir obter um sistema de simulação de código ST foi necessário o desenvolvimento de alguns projetos de programação em paralelo. A utilização do *software* de ambiente de desenvolvimento *Visual Studio* veio permitir a ponte e interligação entre os vários projetos envolvidos.

A estrutura deste sistema de compilação e simulação da linguagem de texto estruturado pode ser organizada e estruturada pelos seguintes elementos:

- Compilador *Lex & Yacc* ;
- Interpretador de código ST (`Interpreter_ST`);
- DLL com o *Parser* (`Parse_DLL`);
- Simulador através de Interface (GUI).

Como já foi referido, o reconhecedor baseado em *Lex & Yacc* é composto pelos dois *scripts* referentes à estrutura léxica e à estrutura sintática e gramatical, respetivamente.

O ficheiro "`thesis.l`" correspondente à configuração léxica do código, contém todas as regras que vão fazer corresponder *strings* lidas de uma entrada *input* (escrita do teclado, leitura de um ficheiro, leitura da interface, etc), e vão converter essas *strings* em *tokens*. Os *tokens* são as representações numéricas das *strings*, que vão simplificar o processo (detalhe em secção 3.2.4). O *Lex* encontra esses identificadores vindos do *stream* de entrada, e coloca-os numa tabela de símbolos. Todas as referências para esses identificadores referem-se ao índice da tabela de símbolos apropriada.

O *Yacc* gera código C para um analisador sintático (*parser*), utilizando regras gramaticais que permitem analisar *tokens* a partir do *lex*, e criar uma árvore sintática (*syntax tree*), impondo uma estrutura hierárquica aos *tokens*. O ficheiro "`thesis.y`" criado contém todas as descrições gramaticais necessárias, neste caso do projeto, com as regras gramaticais referentes à linguagem de programação de texto estruturado (ST), cujo *Yacc* vai ler e gerar o analisador sintático (*parser*).

O interpretador de código de texto estruturado (ST) constitui todas as funções reconhecidas anteriormente pelo analisador sintático, e interpreta-as através dum conjunto de instruções e funções programadas em C. Estas funções a executar variam consoante o tipo de *tokens* que foram identificados e a árvore sintática construída. Em linguagem ST existem muitas instruções condicionais (`IF, ELSE`, etc), temporizadores, funções lógicas, entre outras, e consoante os *tokens* e a relação destes com as respetivas instruções são definidas funções para o interpretador poder saber o que vai e como vai executar.

O interpretador de ST ("`Interpreter_ST`") é composto por dois ficheiros de código em C, onde são definidas e implementadas as funções utilizadas a executar a correspondente gramática definida pelo *Yacc* e enviadas ao analisador sintático (foi especificado com mais detalhe na Secção 3.3.3). Os ficheiros são:

- "`interpreter.c`": contém a função a executar, perante a operação desejada e definida pela gramática;
- "`node_routines.c`": contém outras funções em C adicionais, maioritariamente necessárias na definição, utilização e configuração de temporizadores.

O simulador a partir da interface gráfica foi desenvolvido em C#. Constitui toda a janela de interação direta com o utilizador. Contém botões para ativar as várias funcionalidades, como salvar (*Save*), carregar (*load*) ou correr/parar (*Run/Stop*) o programa. Apresenta também os botões ou interruptores relativos aos endereços de entradas (*inputs*) com os respetivos *LED's* de sinalização, e apenas *LED's* de sinalização relativos aos endereços de saída (*outputs*).

A interface corre o código ST através duma caixa de texto, onde o utilizador pode escrever o respetivo *script* de programação ou importá-lo (*Load*). Tendo em conta os objetivos propostos pelo *kit* de simulação SML (Figura 3.7), a utilização de temporizadores é imprescindível, e portanto a estrutura da interface apresenta também a informação corrente dos temporizadores, ou até outros valores interessantes como o número de *statements* utilizados ou o valor lógico de outra qualquer variável.

É possível observar a estrutura da interface gráfica na Figura 3.22.

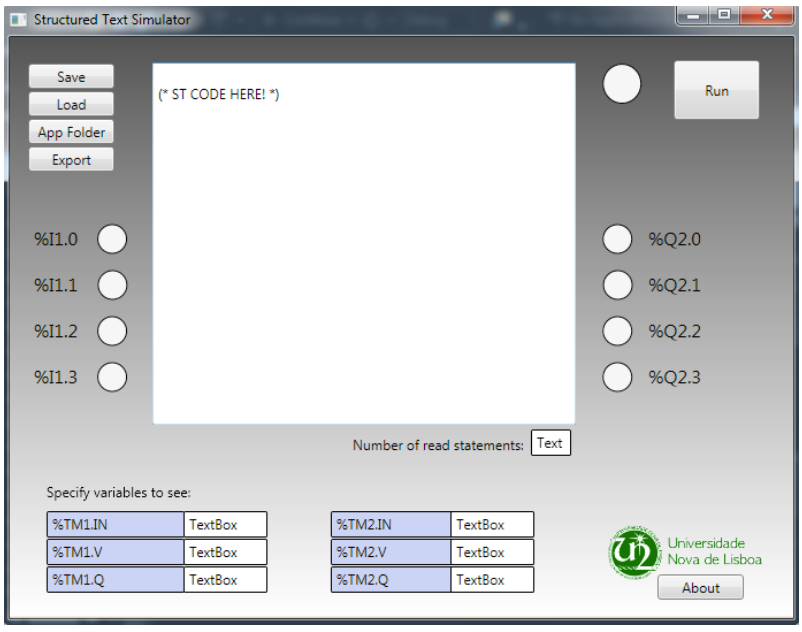


Figura 3.22: Estrutura da interface gráfica de utilizador.

O ficheiro de programação da interface é composto por várias funções e secções, que permitem o ciclo de funcionamento do simulador e as atualizações dos respetivos valores de memória ou de endereço das várias variáveis.

A Tabela 3.8 apresenta as principais secções de código realizado em C# para a construção da interface "`PL7_simulator`".

Tabela 3.8: Organização da interface gráfica.

Secções	**Funcionalidade**
Declarações DLL	Lista de funções importadas pela DLL a serem utilizadas posteriormente por funções locais ou métodos.
Configuração de Botões	Especifica e configura as ações dos botões *Run/Stop, Save, Load*
(Start/Stop) Timer	Cria um sistema de filas de itens de trabalho (*dispatcher*), cria um evento *Tick*, define intervalo de tempo do *dispatcher*
Configuração do *Tick*	A cada ciclo do *dispatcher*, executa um conjnto de instruções e funções
Update dos *Imputs*	Verifica o estado de cada entrada e consoante o seu valor lógico, altera também a cor do *Led*

A realização de uma interface gráfica em C# através do sistema *Windows* (*Microsoft Visual Studio (VS)*) causou a necessidade da utilização de DLL (*Dynamic Link Library*), uma vez que, o suporte de *GUI Windows* em C++ para correr neste sistema, traz alguns problemas na sua realização. Com esta adversidade, e como é necessário executar funções feitas em C a partir de código feito em C#, a única alternativa possível é compilar as funções em C para uma DLL. Desta forma, o programa em C# consegue chamar as funções em C dentro da DLL.

Dentro do projeto em C# define-se o caminho para algumas funções dentro da DLL, ou seja, existe apenas um único ficheiro ou DLL (`PL7_parser.dll`) que contém as diversas funções referenciadas em C# (funções em C com o modificador *export*). Os três Projetos ("`Interpreter_ST`", "`PL7_parser`" (para a DLL) e "`PL7_simulator`") reutilizam todos o mesmo código que está no ficheiro "`interpreter.c`", como por exemplo a função "`run_st_code`", onde cada alteração nesse ficheiro afeta os três projetos referidos.

A configuração das ações dos botões de correr ou parar (*Run/Stop*) é realizada através da mesma função de controlo, tendo em conta que, na prática, o botão é o mesmo. Deste modo, quando o utilizador carrega em "`Run`", o conteúdo do seu texto muda para "*Stop*", ficando à espera ser carregado enquanto o simulador trabalha.

Quando o botão é carregado e o programa não está a correr (clicado o botão "`Stop`") é chamada a função importada na DLL (`parse_structured_text(message)`) que funciona similarmente a uma função *main* no qual o argumento "`message`" corresponde ao

código ST escrito na interface. É chamado também a função "`yyparse()`" e inicializado o *buffer*.

No caso de ser pressionado o botão em modo "*Stop*", o texto escrito no mesmo passa a ser "*Run*" e no fim das operações todos os *statements* lidos são libertados. O código relativo a esta configuração dos botões está especificado na Listagem 3.23.

```
private void Button_Click(object sender, RoutedEventArgs e)
        {
            if (!is_running)
            {
                string message = textbox1.Text;
                parse_structured_text(message);
                StartTimer();
                runStopButton.Content = "STOP";
                is_running = true;
            }
            else
            {
                StopTimer();
                runStopButton.Content = "Run";
                is_running = false;

                //libertar todos os statements lidos
                free_st_code();
            }
        }
```

Listagem 3.23: Configuração dos botões "Start/Stop" da interface gráfica.

Assim que o programa começa a correr é criado um sistema de filas de itens de trabalho (*Dispatcher*), determinado o intervalo de tempo para o evento, e definido o evento como um *Tick*. O *Tick* vai corresponder ao processo pelo qual cada ciclo do *Dispatcher*, executa um conjunto de instruções ou funções. O *Tick* é configurado através da função "`Each_Tick`", explicado mais adiante. A configuração da criação do *Dispatcher* pode ser vista na Listagem 3.24.

No caso do programa ser interrompido (carregado o botão "`Stop`"), o *Dispatcher* é colocado como nulo novamente (definido no "`StopTimer()`").

A função criada "`Each_Tick`" corresponde ao evento originado no *Dispatcher*, em particular pelo *Tick* definido quando criado o "`myDispatcherTime`" em "`StartTimer()`". Esta função "`Each_Tick`" invoca e executa todos os conjuntos de instruções e funções correspondentes à atualização de estados das variáveis utilizadas na interface (*inputs, outputs, timers*) ou informação de texto que necessita de atualização a cada ciclo do *Dispatcher*.

Um dos objetivos desta função é criar os *LEDs* de sinalização, gerando elipses e correspondendo cada elipse a uma posição de um vetor. Essas posições correspondem ao

```
public void StartTimer()
{
   if (myDispatcherTimer==null)
   {
      myDispatcherTimer=new System.Windows.Threading.DispatcherTimer();
      myDispatcherTimer.Interval=new TimeSpan(0, 0, 0, 0, 100); // 100
          Milliseconds
      myDispatcherTimer.Tick+=new EventHandler(Each_Tick);
   }
   myDispatcherTimer.Start();
}
```

Listagem 3.24: Definição e criação do "Dispatcher" para o processamento e atualização de dados na interface gráfica.

índice de cada variável de entrada ou valor de saída.

Para atualizar as variáveis de entrada é percorrido o vetor de entradas (`inputs[]`) e para cada posição é correspondido o respetivo índice no vetor de entradas. A função chamada pelo "`Each_Tick()`" responsável pela atualização das variáveis de entrada, ou seja, atribuir as entradas (*inputs*) às correspondentes variáveis, está especificada na Listagem 3.25.

```
public void get_inputs_from_gui_controls()
{
   //colocar inputs nas variaveis
   for (int i = 0; i < inputs.Length; i++)
   {
      set_sym_value(inputs[i], input_values[i]);
   }
}
```

Listagem 3.25: Função que atribui as entradas (*inputs*) às respetivas variáveis na interface gráfica.

Para atualizar as saídas (*outputs*), tendo em conta que, apenas existe sinalização dos *LEDs* e não interação com o utilizador é percorrido o vetor de saídas (`outputs[]`) e comparando o valor de cada posição, o *LED* fica vermelho ou cinzento consoante o seu valor lógico ser 1 ou 0, respetivamente. A atualização dos campos de texto com as informações adicionais ou dos temporizadores é conseguida bastando apenas corresponder o texto respetivo ao seu valor lógico correspondente.

Fora da função "`Each_Tick`"são atualizados também os *LEDs* inerentes aos valores lógicos das entradas. Esta funcionalidade está fora do evento, tendo em conta que depende da interação manual do utilizador.

As configurações dos botões de *Load* e *Save* permitem definir o modo de carregamento do ficheiro com o código de texto estruturado (*import file*), ou gravar o código escrito na interface num ficheiro.

CAPÍTULO 4

RESULTADOS EXPERIMENTAIS

Os resultados experimentais visam ilustrar as experiências realizadas que comprovam os objetivos iniciais propostos para o projeto.
Este capítulo está dividido em duas secções principais:

- 4.1 - Testes preliminares e simulações;
- 4.2 - Resultados obtidos.

Na secção 4.1 são ilustrados e especificados todos os testes preliminares realizados, como modelo a atingir para os resultados finais desejados. Tendo em conta que o objetivo proposto é simular, por meio de uma interface gráfica com o utilizador (*GUI*), o funcionamento do *kit* SML (*Washing Machine*), foi necessário realizar um código em ST (*structured text*) que funcione no *kit* e que cumpra os requisitos. Utilizando este código testado previamente na máquina são ilustrados os respetivos testes nesta secção.

A interface gráfica realizada necessitou de alguns testes prévios, antes de simular os requisitos propostos pelo *kit*. Nesta secção também são descritos alguns testes simples de simulação na interface com instruções em código ST.

Na secção 4.2 todos os resultados obtidos são apresentados. Como na secção (4.1), os testes no *kit* já são realizados, apenas são especificados os resultados obtidos no simulador criado em C# com o código ST analogamente testado diretamente na máquina.

Na validação de resultados são justificados os respetivos resultados obtidos nas várias experiências.

4.1 Testes Preliminares e Simulações

4.1.1 Teste no Kit SML "*Washing Machine*" do Código ST

O teste no *kit* SML com o código ST corresponde ao teste que permite construir um modelo de código, a ser seguido na posterior construção da interface gráfica.

Relembrando as várias etapas propostas pelo projeto utilizando o *kit* SML (secção 3.2.2), o objetivo é:

1º) arrancar a máquina ativando o botão "`%I1.6`", enchendo a cuba até ao nível "H2" (sensor `%I1.3`) por ação de "IN" (motor `%Q2.3`);

2º) depois de atingir o nível "H2" deve acionar o motor "M" por ação de "N2" (`%Q2.0`) durante 10 segundos;

3º) por fim, a cuba esvazia por ação de "OUT" (`%Q2.4`) durante 15 segundos.

Inicialmente todos os processos estão parados e, para haver um enquadramento do *kit* com os respetivos sensores e *Leds* correspondentes é mostrada a Figura 4.1.

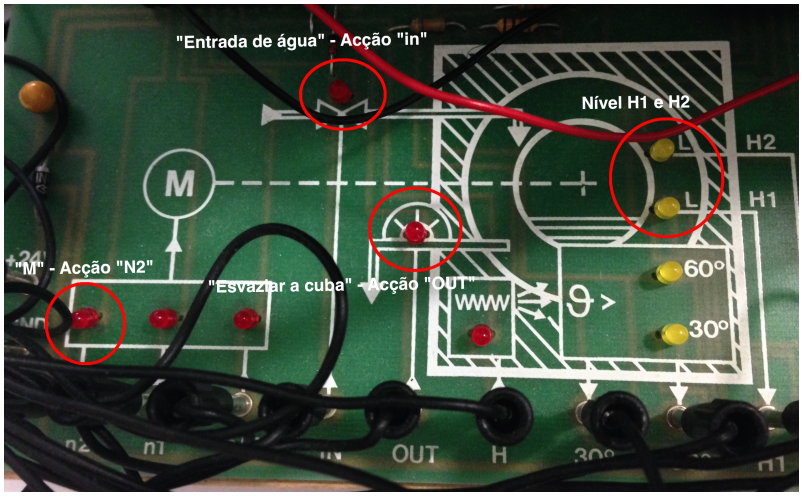
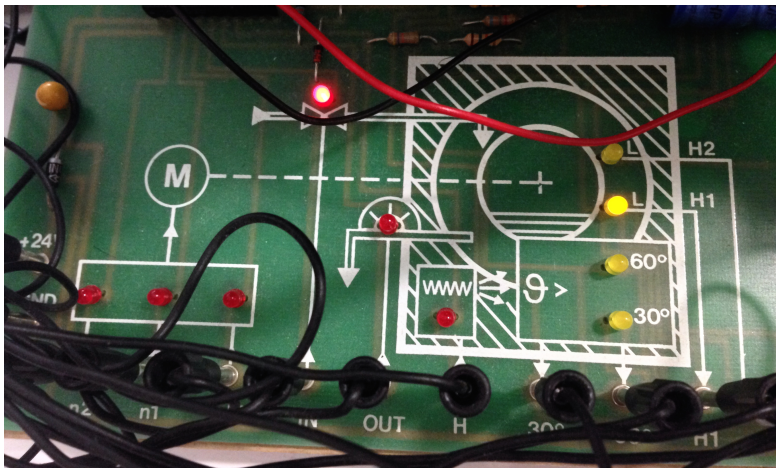


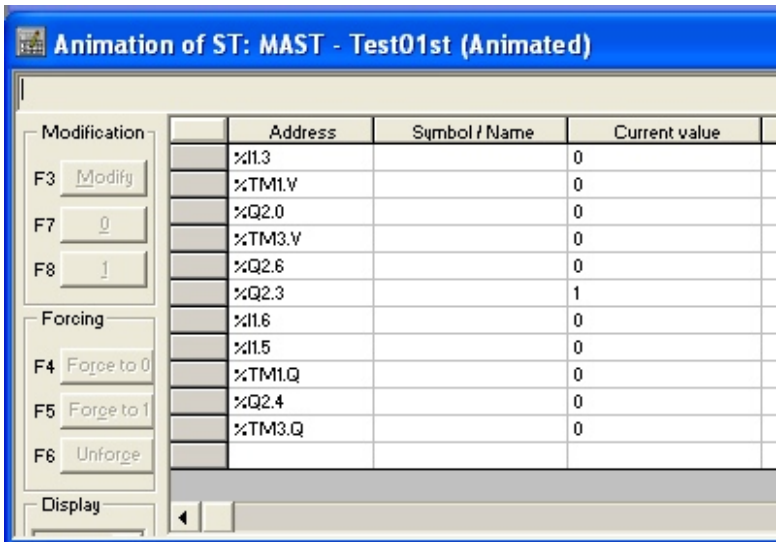
Figura 4.1: Aspeto inicial do *kit* SML para autómato TSX3721 e respetiva sinalização dos *leds* correspondentes às ações ou sensores utilizados.

O primeiro passo foi ligar o sistema através do botão "*ON / OFF*", ou seja, o botão com a entrada correspondente "`%I1.6`". A Figura 4.2 mostra a cuba a encher, como previsto, e o nível de água a subir (neste caso está no nível "H1").

Pode-se ver a relação dos *bits* (Figura 4.2b) com os *LEDs* acesos (Figura 4.2a) , onde nesta etapa a ação "IN" está a decorrer (ver *led* correspondente na Figura 4.1), representando a cuba a encher, e o nível da água representado corresponde, neste estágio, ao nível H1.



(a) *Kit* SML com os *leds* correspondentes (ação IN e nível H1).

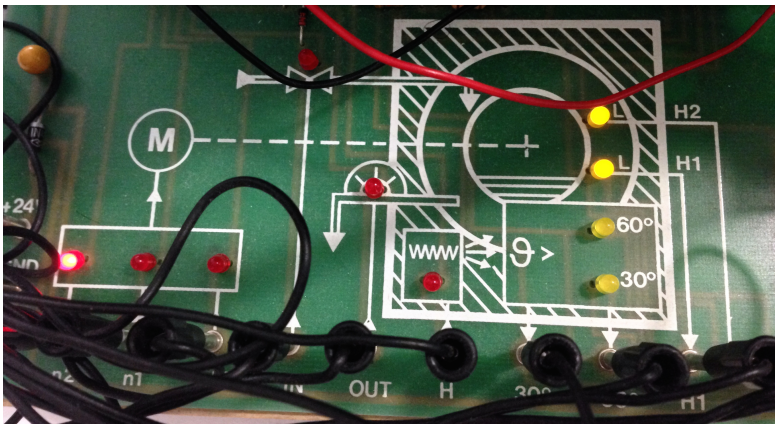


(b) Tabela de valores (no PL7) dos correspondentes endereços das variáveis de entrada, saída e temporizadores (ação IN - %Q2.3).

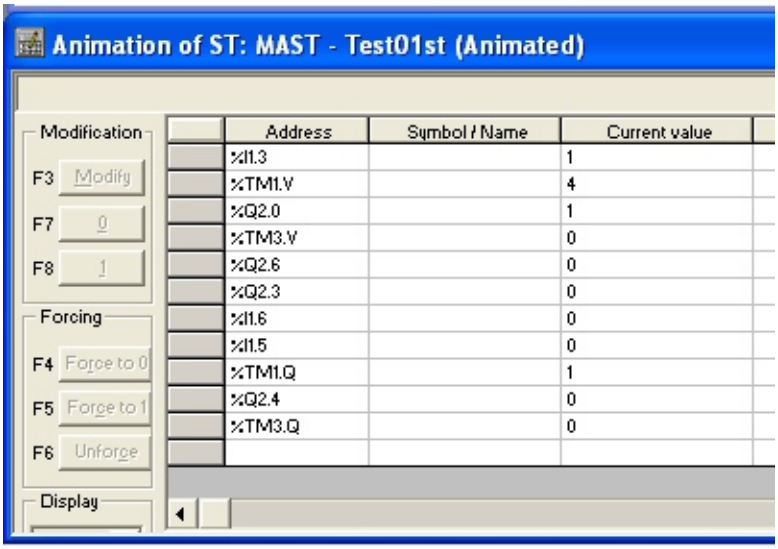
Figura 4.2: Comportamento do *kit* SML com a cuba a encher e nível H1 estabelecido.

Quando é atingido o segundo nível (H2) a cuba deixa de encher e, logo de seguida, é iniciado o funcionamento do motor (M) por ação de "N2". Este funcionamento tem uma duração de 10 segundos.

A Figura 4.3 apresenta os resultados experimentais do comportamento descrito, onde na Figura 4.3a se pode observar o nível "H2" atingido e consequentemente o funcionamento do motor (M) e respetiva ação "N2". Na Figura 4.3b é visível o valor dos endereços associados, com o sensor "`%I1.3`" do nível H2 ligado, o endereço do motor (M) "`%Q2.0`" ligado, e o funcionamento em progressão, neste caso com 4 segundos de tempo percorrido, do temporizador (TM1) correspondente aos 10 segundos.



(a) *Kit* SML com os *leds* correspondentes (ação N2 e nível H2).

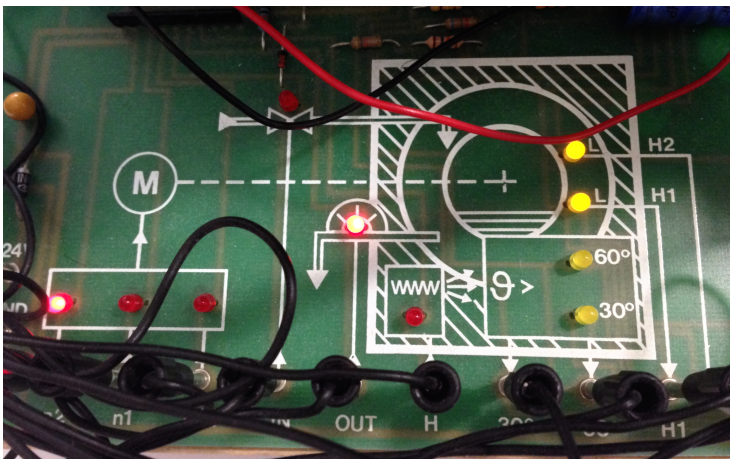


(b) Tabela de valores (no PL7) dos correspondentes endereços das variáveis de entrada, saída e temporizadores (Ação N2 - %Q2.0, nível H2 - %I1.3).

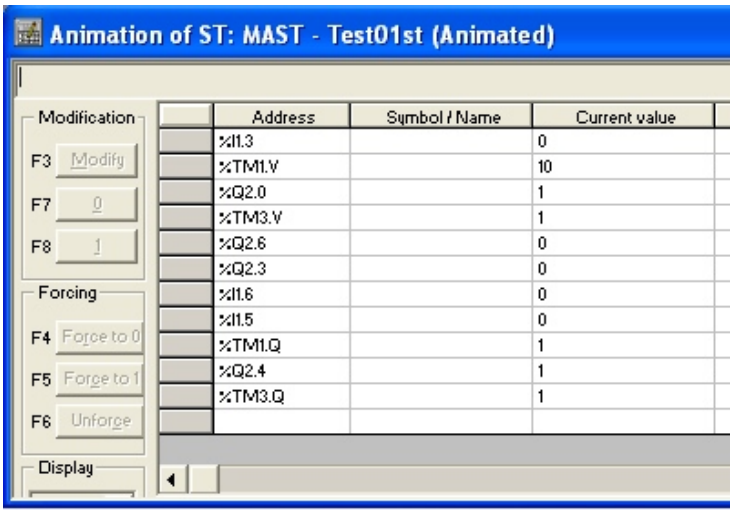
Figura 4.3: Comportamento do *kit* SML com o motor (M) em funcionamento e nível H2 atingido.

Após os 10 segundos de funcionamento do motor (M), e respetiva ação "N2" é ativado o esvaziamento da cuba através da ação "OUT". Esta nova ação está em funcionamento por 15 segundos, tempo suficiente para o nível da água descer.

A Figura 4.4 elucida o momento de passagem entre a ação "N2" correspondente ao funcionamento do motor (M) e a ação "OUT" correspondente ao inicio do esvaziamento da cuba. Na Figura 4.4a é possível verificar o acionamento da saída de água da cuba. Como a mudança de ação é sequencial e contínua, ainda se verifica o funcionamento do motor e o esvaziamento da cuba, mas apenas por um instante. Na Figura 4.4b verifica-se que já passaram os 10 segundos do *Timer* "`%TM1`" e que o novo (`%TM3`) acabou de iniciar (com 1 segundo percorrido apenas). Pelo endereço "`%Q2.4`", verifica-se que o esvaziamento da cuba está a ocorrer (respetiva ação OUT).



(a) *Kit* SML com os *leds* correspondentes (ação OUT, fim de ação N2, e nível H2).

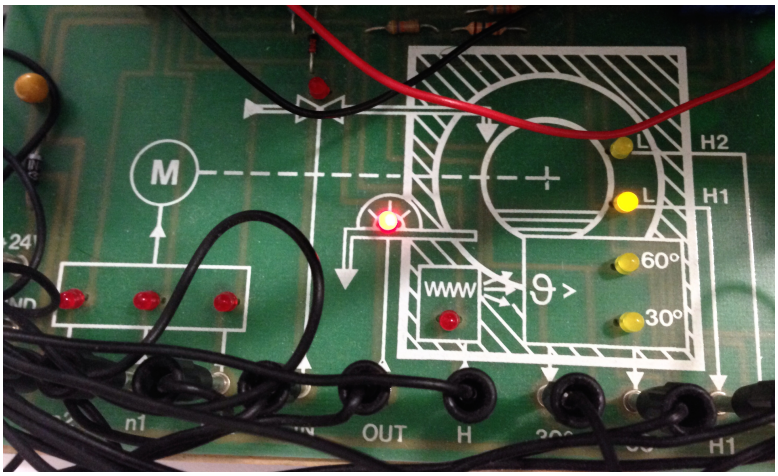


(b) Tabela de valores (no PL7) dos correspondentes endereços das variáveis de entrada, saída e temporizadores (Ação N2 - %Q2.0, nível H2 - %I1.3, ação OUT - %Q2.4).

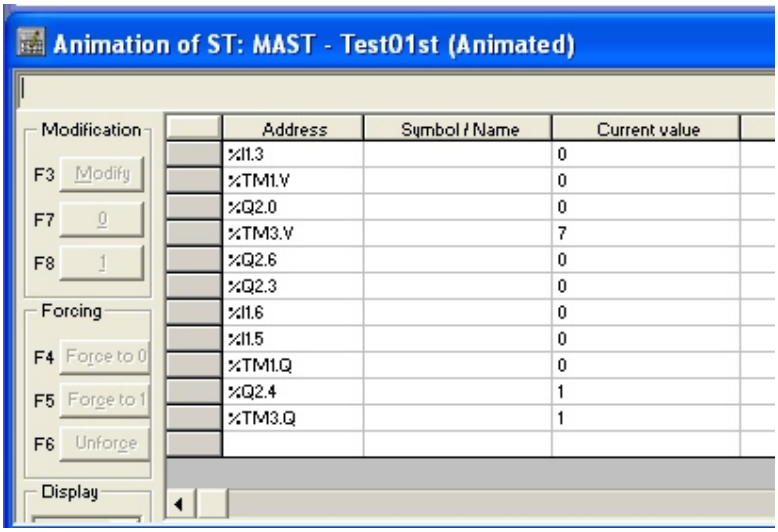
Figura 4.4: Comportamento do *kit* SML com o início do esvaziamento da cuba após 10 segundos de funcionamento do motor M.

À medida que a ação "OUT" vai ocorrendo, os níveis de água na cuba vão diminuindo.

Quando o esvaziamento da cuba está em progressão, apenas a ação "OUT" está a ocorrer (durante os 15 segundos previstos) e os níveis de água vão diminuindo, como se pode observar pela Figura 4.5a, onde o nível já desceu para o H1. Este processo ocorre com aproximadamente 7 segundos de funcionamento (verificar valor do *timer* `%TM3` na tabela, Figura 4.5b), e nesta altura, pode-se verificar também que mais nenhuma ação está a ocorrer de momento.



(a) *Kit* SML com os *leds* correspondentes (ação OUT e nível H1).

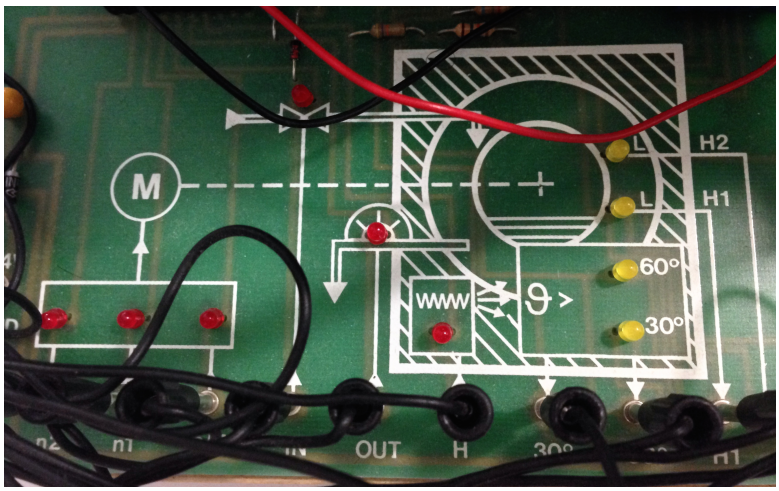


(b) Tabela de valores (no PL7) dos correspondentes endereços das variáveis de entrada, saída e temporizadores (ação OUT - %Q2.4, *Timer* TM3 activo - %TM3.Q:=1).

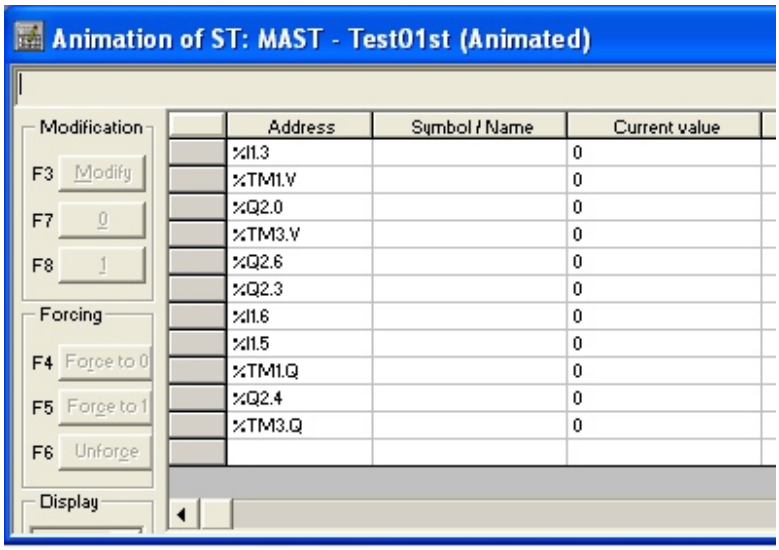
Figura 4.5: Comportamento do *kit* SML com esvaziamento da cuba em progressão.

Quando os 15 segundos terminarem, a cuba esvaziou na totalidade, e a ação "OUT" é desativada de imediato.

Como se pode observar pelas duas Figuras (4.6a, 4.6b), após o esvaziamento da cuba, ou seja, os 15 segundos de funcionamento da ação "OUT", o nível de água desceu abaixo do nível H1, e a respetiva ação é desativada, tornando o sistema ao estado inicial novamente.



(a) *Kit* SML com os *leds* correspondentes (estado finalizado).



(b) Tabela de valores (no PL7) dos correspondentes endereços das variáveis de entrada, saída e temporizadores (retorno ao estado incial).

Figura 4.6: Comportamento do *kit* SML com finalização de processos.

4.1.2 Testes e Exemplos Preliminares do Compilador em Lex e Yacc

Na secção da implementação do compilador em Lex e Yacc (secção 3.3.3) vimos que para se criar um executável capaz de interpretar e correr texto estruturado é necessário compilar os ficheiros que contêm o analisador léxico (ficheiro `lex.yy.c`), o *parser* (ficheiro `thesis.tab.c`), e todos os ficheiros com as funções e estruturas desenvolvidas para a construção do compilador e interpretador (`node_routines.c` e `interpreter.c`).

A Figura 4.7 ilustra o momento de criação do executável "`thesis`", capaz de executar as instruções e interpretar os vários *statements* escritos no código.

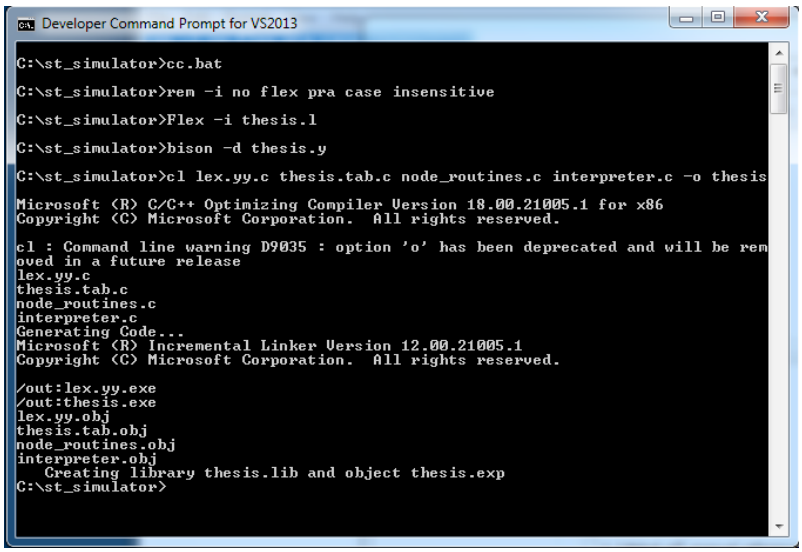


Figura 4.7: Criação do ficheiro executável a partir da compilação Lex e Yacc.

Uma vez criado o ficheiro executável com sucesso, foi necessário testar e provar que as definições e produções gramaticais realizadas na construção do compilador baseado em Lex e Yacc, foram concebidas corretamente.

Foram realizados um conjunto de testes utilizando as principais instruções da linguagem de texto estruturado.

Como o objetivo inicial de teste era simular o comportamento e desempenho do *kit* SML (*Washing Machine*) utilizado no autómato TSX 3721 e, obtendo o código ST testado previamente e diretamente na máquina como modelo, as principais instruções desta linguagem a utilizar são os operadores condicionais (`IF`, `ELSIF`, `ELSE`), os operadores lógicos (`OR`, `AND`),e a utilização de temporizadores.

Correndo o executável "`thesis`", o primeiro teste a realizar foi precisamente o dos operadores condicionais. Na verdade e observando a Figura 4.8, o código que realmente interessa testar é o bloco referente à estrutura do `IF`. As inicializações das variáveis de

entrada (`%I1.0` e `%I1.1`) e das instruções de impressão (`PRINT`) são apenas código necessário para o teste e não é para ser contabilizado como instrução de texto estruturado.

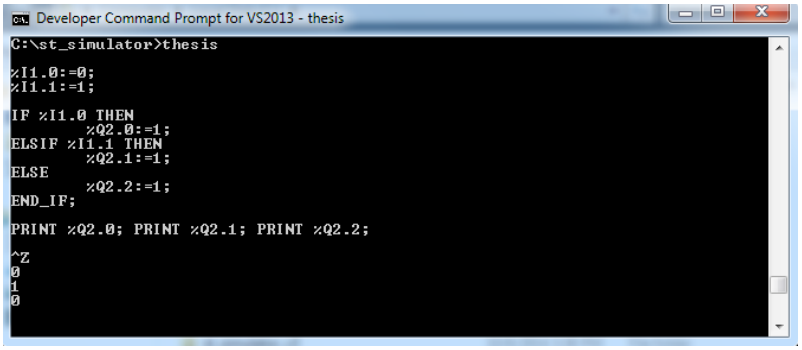


Figura 4.8: Teste das instruções condicionais (*IF, ELSIF, ELSE*) na linha de comandos.

Verificando o resultado final (informação que se segue à instrução de finalização da linha de comandos, `^Z`) da Figura 4.8, verifica-se que que as operações ocorrem com sucesso. Apenas o endereço `%I1.0` tem o valor lógico 1. As condições especificadas previam que caso esse endereço tivesse o valor lógico 1, então o endereço de saída `%Q2.0` estaria a 1 também e consequentemente, o endereço `%Q2.2` estaria a 0 (valor lógico 1 atribuído apenas como condição `ELSE`). A variável de saída `%Q2.1` apenas estaria com o valor lógico 1, caso a correspondente variável de entrada `%I1.1`, também tivesse esse valor.

O próximo teste em estudo e experimentado é o teste relativo à utilização dos operadores lógicos em linguagem de texto estruturado.

O ensaio correspondente à Figura 4.9a da Figura 4.9, tem um conjunto de 4 variáveis de entrada de endereços:

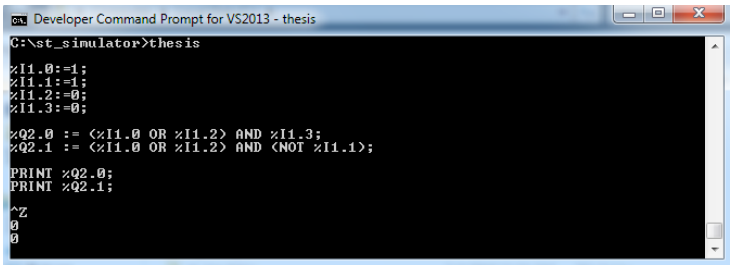
```
%I1.0 := 1;
%I1.1 := 1;
%I1.2 := 0;
%I1.3 := 0;
```

A primeira instrução prevê que a saída `%Q2.0` tenha o valor lógico 1, caso existam duas variáveis de entrada em simultâneo com o respetivo valor lógico 1. Neste exemplo, seria em simultâneo juntamente com a variável de endereço `%I1.3`, a variável `%I1.0` ou `%I1.2`. A segunda instrução com os operadores lógicos tem um comportamento análogo ao anterior, apenas referenciando que em simultâneo às mesmas variáveis `%I1.0` ou `%I1.2`, a variável de endereço `%I1.1`, não pode conter o valor lógico 1.

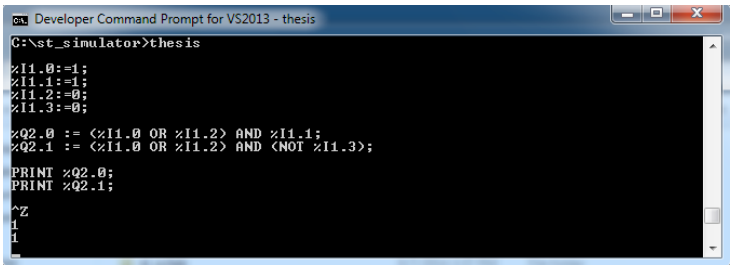
O segundo exemplo correspondente à Figura 4.9 (4.9b) apresenta uma estrutura de teste igual ao exemplo anterior, apenas tendo como objetivo que o resultado das saídas `%Q2.0` e `%Q2.1` tenham igualmente o resultado lógico 1.

Os valores associados às variáveis de entrada, comparando com o primeiro exemplo

são os mesmos. Assim sendo, a única alteração recai nas instruções envolvendo os operadores lógicos. É possível observar que este segundo exemplo mantém a estrutura das instruções, trocando apenas as variáveis de entrada na ação dos operadores lógicos `AND` e `AND NOT`, ou seja, trocando a variável `%I1.1` pela `%I1.3` e vice-versa, comparando mais uma vez com o exemplo anterior.



(a) Teste com resultado lógico 0.



(b) Teste com resultado lógico 1.

Figura 4.9: Teste dos operadores lógicos (*OR, AND, AND NOT*) na linha de comandos.

Analisando os resultados obtidos do primeiro caso (Figura 4.9a), verificamos que o resultado obtido das saídas `%Q2.0` e `%Q2.1` são igualmente 0. O resultado está correto, uma vez que a variável de entrada `%I1.3` tem o valor lógico 0, e com o operador lógico `AND` associado, o resultado só podia ser 0 também. Analogamente, podemos verificar que a segunda instrução associada à saída `%Q2.1` também poderia apenas conter o valor lógico 0, tendo em conta a existência do operador `AND NOT` associado à entrada `%I1.1`, entrada esta que contém o respetivo valor 1.

No segundo caso (Figura 4.9b), o resultado obtido por cada uma das saídas é igualmente 1. Podemos concluir que o resultado está correto uma vez mais porque a saída `%Q2.0` estaria a 1 se as entradas `%I1.0` ou `%I1.2` estivessem a 1, juntamente com a entrada `%I1.1`. Podemos verificar pela Figura 4.9b a confirmação dos valores de entrada. A segunda instrução tal como na Figura 4.9a, tem um operador "`AND NOT`", mas como o valor da variável de entrada `%I1.3` está a 0 e `%I1.0` está a 1, o resultado da operação é 1.

A Figura 4.10 retrata o teste referente ao comportamento de um temporizador *timer*

TP (*Pulse Timer*). O objetivo deste tipo de temporizadores é permitir que, durante o intervalo de tempo pré-definido, a sua saída correspondente (`Q`) esteja ativa.

Com um tempo base de 1 segundo, e uma duração de aproximadamente 10s é definido o temporizador `%TM1`. O temporizador é iniciado (`START %TM1;`) quando a variável de entrada `%I1.0` estiver ativa. Durante esse tempo e como a variável de saída `%Q2.0` está associada à saída do temporizador, então, tem também o valor lógico 1.

Para se conseguir observar o comportamento do temporizador na linha de comandos, realizou-se um ciclo *While* que durante os 10 segundos aproximados, vai imprimindo o valor da variável de saída e do respetivo segundo a que o temporizador está a contar no momento.

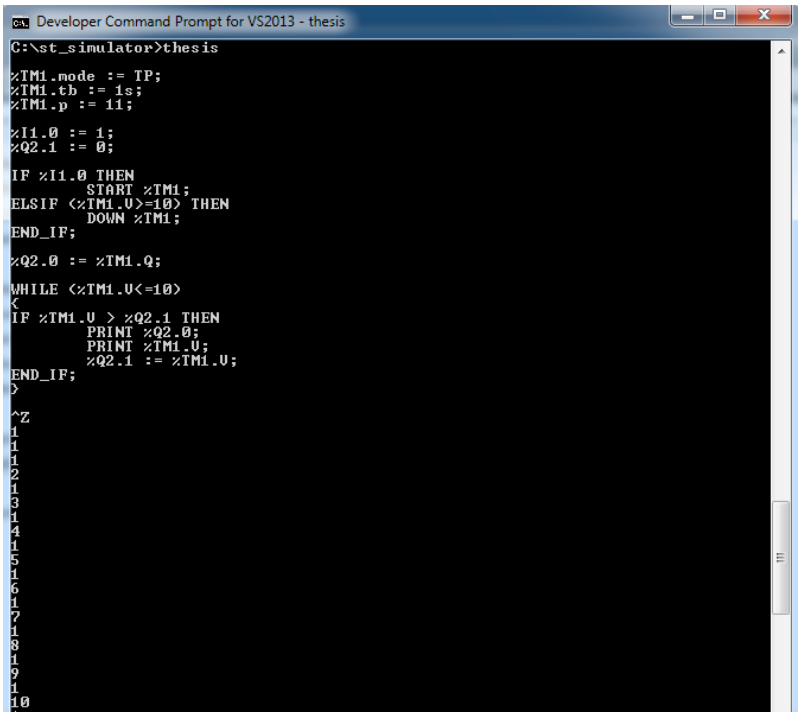


Figura 4.10: Teste de um temporizador TP na linha de comandos

O resultado ilustrado, mostra que durante os 10 segundos de funcionamento a variável de saída tem sempre o valor lógico 1 associado. Ao fim de 10 segundos de funcionamento (`%TM1.V>=10`), o temporizador é desativado (`DOWN %TM1;`).

4.2 Resultados Obtidos

4.2.1 Simulação da Interface Gráfica com Código ST a Testar no *Kit*

O estado inicial do simulador contém todas as variáveis de entrada e saída com o respetivo valor lógico a 0. Os valores dos temporizadores também estão igualmente inicializados e parados. A Figura 4.11 ilustra o estado inicial das variáveis de entrada e saída, e os valores relativos aos temporizadores utilizados.

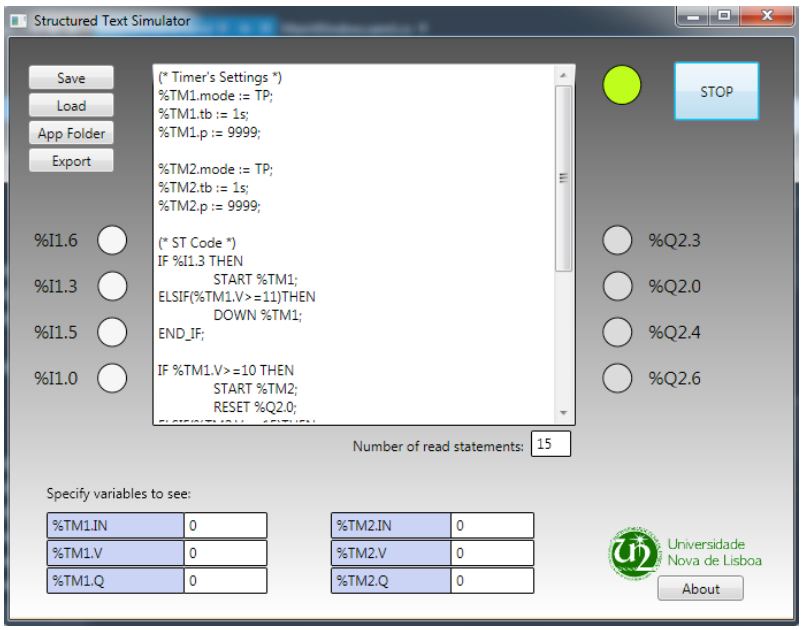


Figura 4.11: Aspeto inicial da interface gráfica de utilizador.

O código em linguagem de texto estruturado utilizado experimentalmente para obtenção de resultados, como já foi referenciado, foi realizado com o objetivo de corresponder ao comportamento do *kit* de máquina de lavar (SML) para o autómato TSX3721.

O primeiro passo do processo corresponde à simulação de entrada de água na cuba (ação IN - `%Q2.3`), onde podemos ver o botão de arranque de sistema (*ON* - `%I1.6`) ativo, tal como o respetivo *LED* de sinalização (`%Q2.3`) referente ao endereço da variável relativa à ação "IN" (entrada de água na cuba).

A Figura 4.12 mostra que quando o botão de arranque (`%I1.6`) é ligado, a cuba começa a encher de água (*LED* `%Q2.3` ativo). Como o objetivo é representar botões de impulso na entrada, se carregarmos de novo no botão de arranque, ele desliga a luz mas a cuba continua a encher.

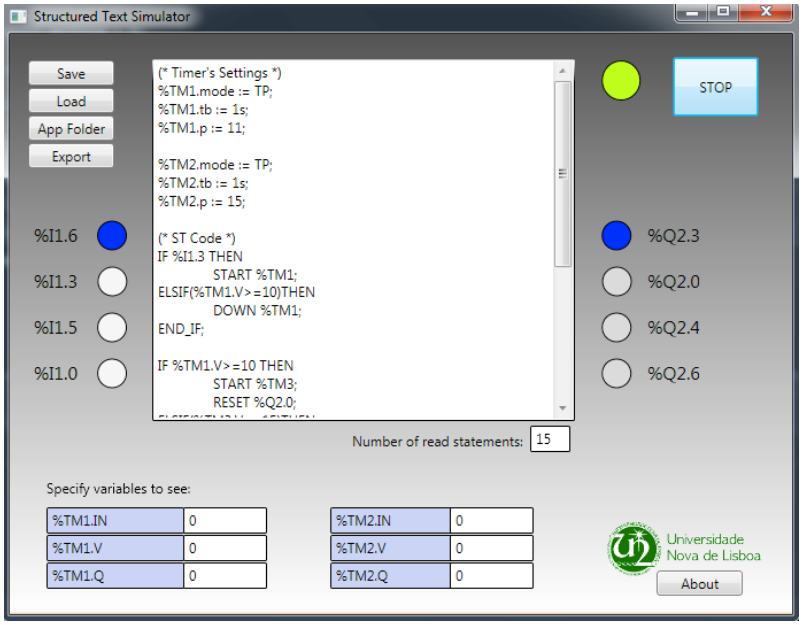


Figura 4.12: Interface gráfica representando o funcionamento da entrada de água na cuba (ação IN - %Q2.3).

Quando a cuba encher de água até ao nível "H2" (sensor `%I1.3`), deverá parar, e iniciará o funcionamento de um motor (M) por ação de "N2". Como a interface representa um simulador, a indicação por parte do sensor de nível "H2", tem de ser realizado manualmente, bastando apenas carregar no botão correspondente à variável `%I1.3`, como se pode observar na Figura 4.13.

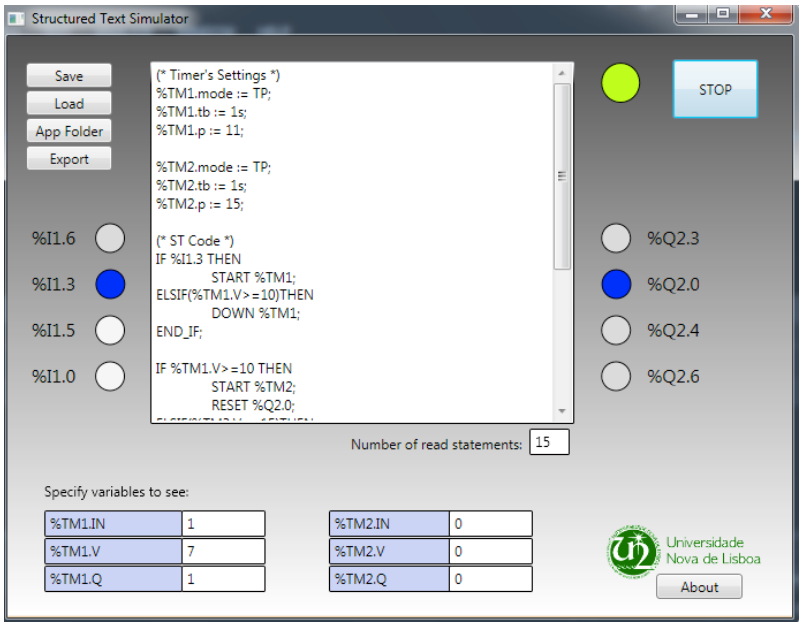


Figura 4.13: Interface gráfica representando o funcionamento do motor (M) (ação N2 - %Q2.0).

Quando o sensor é ativado, o motor (M) é iniciado e a ação de entrada na água é interrompida, como se pode observar na Figura 4.13. O temporizador `%TM1` é iniciado simultaneamente com o funcionamento do motor, para permitir apenas que este trabalhe durante os 10 segundos aproximadamente estipulados. É possível observar que o temporizador `%TM1` está com 7 segundos de tempo percorrido (`%TM1.V=7`).

O funcionamento do motor é de aproximadamente 10 segundos, e portanto, quando este tempo passar é iniciado o esvaziamento da cuba (ação OUT - `%Q2.4`). Na Figura 4.14 observa-se o momento de transição entre uma ação (N2) e outra (OUT). Neste momento, o esvaziamento da cuba iniciou e ainda se está a finalizar o funcionamento do motor (reparar que o temporizador `%TM1` está com 10 segundos contabilizados, e o temporizador `%TM2` acabou de iniciar). O processo de esvaziamento da cuba tem aproximadamente 15 segundos (tempo contabilizado pelo temporizador `%TM2`).

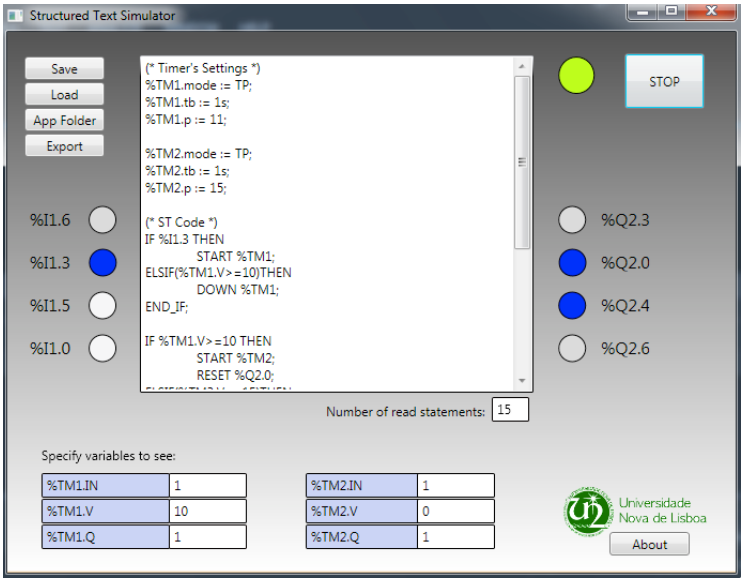


Figura 4.14: Interface gráfica representando o início do esvaziamento da cuba, com a interrupção do motor (M) (início da ação OUT - %Q2.4).

A Figura 4.15 representa somente o processo de esvaziamento da cuba, onde se pode verificar, que o nível de água já não está no nível "H2" (sensor `%I1.3` tem de ser desligado manualmente) e que o funcionamento do motor (ação `%Q2.0`) já não está a decorrer. Em contrapartida, e neste caso com 7 segundos de esvaziamento, a ação "OUT" (`%Q2.4`) é a única que está em funcionamento.

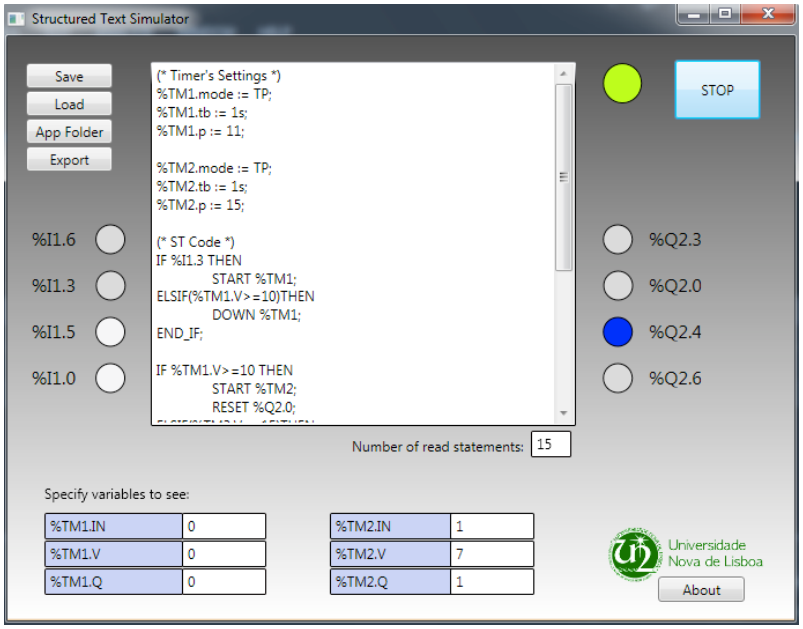


Figura 4.15: Interface gráfica representando o esvaziamento da cuba (ação OUT - %Q2.4).

Por fim, após os 15 segundos de funcionamento do temporizador `%TM2` e respetiva ação "OUT" de esvaziamento da cuba, o estado das variáveis de entrada (botões / sensores) e de saída (ações), volta ao ponto inicial, e pronto a ser utilizado novamente por qualquer utilizador, como ilustra a Figura 4.16.

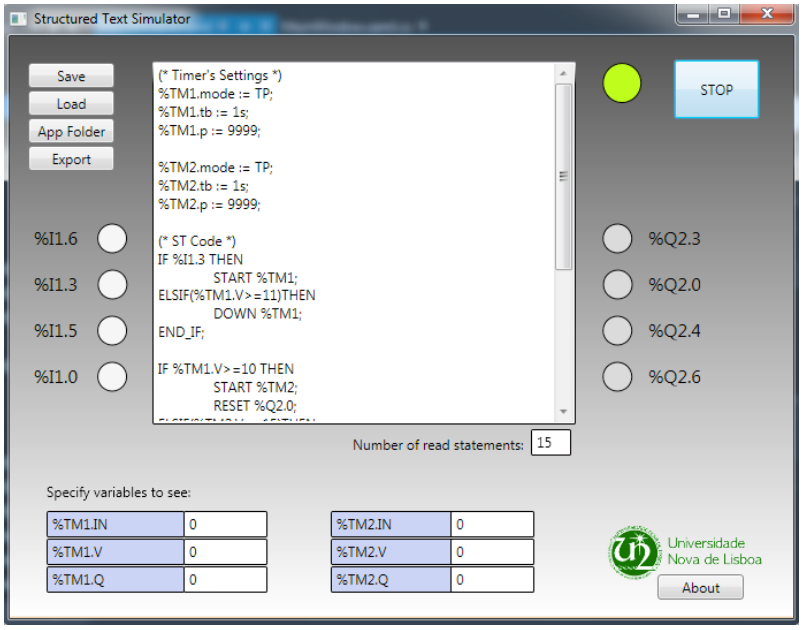


Figura 4.16: Estado final no processo de funcionamento de simulação do *kit* SML na interface gráfica de utilizador.

O simulador também opera um modo de segurança que permite interromper qualquer ação que esteja a decorrer, parando também, o funcionamento de qualquer temporizador. Na Figura 4.17 o interruptor de emergência `%I1.5` está acionado, e como se pode observar o *LED* correspondente à paragem (`%Q2.6`) está ligado. Nenhuma ação ou temporizadores estão em funcionamento, mesmo com o sensor de nível "H2" ativo.

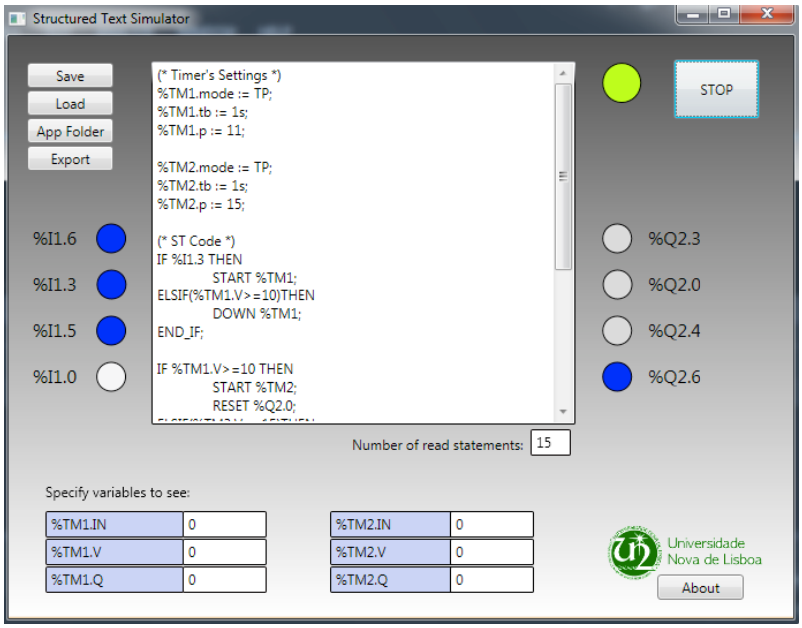


Figura 4.17: Modo de simulação de paragem de emergência na interface gráfica.

Apesar do simulador não indicar explicitamente o local exato da ocorrência de um erro, indica o número de *statements* lidos corretamente no *script* de código escrito, ou seja, indica o número de blocos de instruções reconhecidos corretamente. Esta funcionalidade permite verificar se o código contém algum erro sintático nalguma instrução escrita (vai estar no *statement* com a posição exatamente a seguir ao número lido).

Admitindo que existem 15 *statements* previstos no código de texto estruturado escrito pelo utilizador, se o simulador indicar os 15 *statements* como lidos, então o programa está sintaticamente correto.

A Figura 4.18 indica que foram lidos apenas 8 blocos de instruções, ou seja, que o 9º *statement* está sintaticamente incorreto. Podemos observar que falta o carácter "`;`" depois da instrução "`END_IF`".

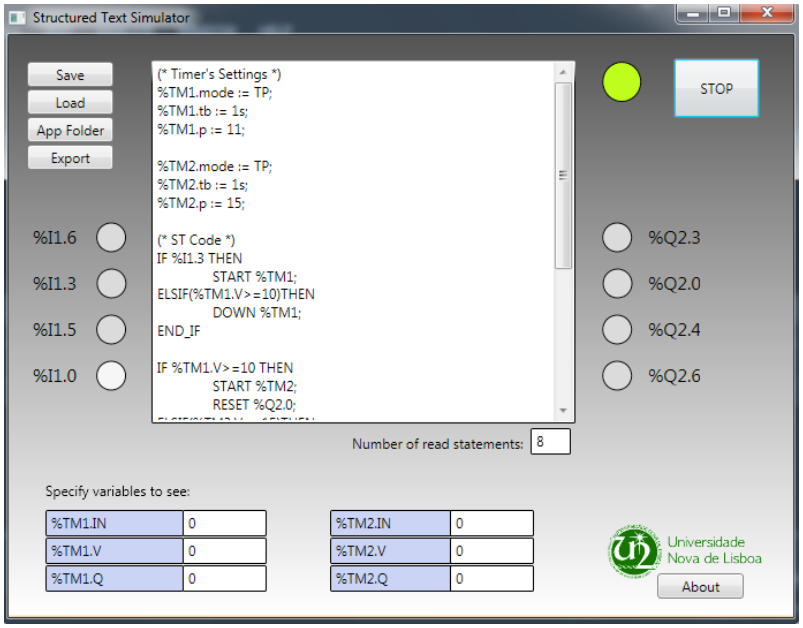


Figura 4.18: Modo de simulação com um erro sintático associado.

4.2.2 Validação de Resultados

Os testes experimentais foram realizados tendo como base três etapas principais de teste:

1) Teste no autómato TSX3721 utilizando o *kit* SML:

   ✓ Atingido os objetivos requeridos para o funcionamento proposto do *kit* utilizado;

   → O código de texto estruturado (ST) utilizado serviu como modelo para a realização do compilador e respetiva interface gráfica.

2) Teste do compilador baseado em Lex/Yacc de código de texto estruturado (ST):

   ✓ Exemplos com as principais instruções (condicionais, atribuições, operadores lógicos e temporizadores) realizados com sucesso;

   → Os testes permitiram confirmar que a estrutura, interpretação e compilação de texto estruturado podiam ser utilizados para a realização da interface gráfica.

3) Teste final na interface gráfica de utilizador (*GUI*):

   ✓ Código do modelo de texto estruturado, previamente utilizado e testado no autómato TSX3721, simulado com sucesso na interface gráfica;

   ✓ Objetivos propostos pelo *kit* SML cumpridos com sucesso com a simulação na interface gráfica.

CAPÍTULO 5

CONCLUSÃO

No capítulo final são apresentadas todas as considerações finais. É realizado um sumário dos procedimentos e resultados importantes na conceção do projeto, destacando-se a importância à realização do simulador para linguagem em texto estruturado, o que se concluiu, que contribuições foram realizadas e que resultados se obtiveram face aos esperados ou propostos inicialmente.

Para desenvolvimento em trabalho futuro são discutidos alguns possíveis projetos ou melhorias a realizar mais tarde, que contribuam significativamente para uma melhoria de desempenho do simulador, ou que sirvam de objetivo para um futuro trabalho.

CONCLUSÃO

5.1 Conclusões

A conceção e desenvolvimento de um sistema de simulação de linguagem em texto estruturado, capaz de oferecer ao utilizador uma interface e realização de testes de vários programas executáveis em autómatos, em particular no PLC TSX Micro 3721, exigiu a execução de um conjunto de etapas. O trabalho desenvolvido foi dividido em duas partes principais, uma referente à programação do autómato através da realização de um código em linguagem em texto estruturado, e outra parte correspondente à construção de um compilador de linguagens de programação baseado na tecnologia de Lex e Yacc.

Uma boa programação, principalmente quando aplicada à utilização de autómatos é fundamental para prevenir a existência de erros grosseiros e avarias em máquinas em ambiente industrial. Tendo em conta a inexistência de qualquer simulador de linguagem em texto estruturado no laboratório onde o projeto foi realizado, e o facto de não ser possível à universidade, num futuro próximo, aceder a este tipo de *software* de simulação para esta linguagem de programação, a principal contribuição com o desenvolvimento do projeto é a construção do simulador com interface gráfica para linguagem em texto estruturado e a sua futura utilização em laboratório para a realização de testes.

O desenvolvimento do simulador e respetiva interface gráfica com o utilizador foi construída com sucesso, baseando-se no conjunto de testes realizados previamente no *kit* utilizado no autómato (secção 4.1). Ao se realizar os devidos testes até se obter um programa funcional em linguagem em texto estruturado e que cumprisse com os requisitos iniciais propostos no *kit* (SML - máquina de lavar), o código implementado serviu como modelo para construção do compilador e respetivos simulador e interface gráfica.

A construção do compilador para linguagem em texto estruturado passou por três etapas principais na sua implementação, nomeadamente três geradores de análise:

1. Gerador de Análise Léxica (*Scanner*);
2. Gerador de Análise Sintática e semântica (*Parser*);
3. Interpretador.

Através da análise léxica do código ST implementado (função do Lex) foi possível identificar e reconhecer cada símbolo característico (*token*) da linguagem (e.g. `IF`, `ELSE`,

variáveis de entrada e saída, operadores, temporizadores, etc.). A análise sintática e semântica (ação do Yacc) permitiu criar uma estrutura hierárquica resultantes das regras gramaticais e de produção que cada *token* enviado da análise léxica formou, quando conjugados com outros *tokens*. O significado semântico das regras de produção geradas foi interpretado, e através de um conjunto de funções realizadas em linguagem C, foram definidas as respetivas instruções a serem executadas consoante esse mesmo significado e objetivo.

É possível assim, projetar e implementar um analisador léxico, uma vez que, as expressões regulares que descrevem uma linguagem de programação são um caso particular do CFG (*Context-free grammar*). Da mesma maneira é possível implementar um analisador sintático através desta gramática, que indica se um determinado *statement* pertence, ou não, à linguagem de programação gerado pelo CFG proposto. Na construção de um tradutor ou compilador é de extrema importância, uma boa definição da CFG, para o desenvolvimento e geração de código de linguagens de programação de alto nível (código C, Scilab, etc) a partir de um programa escrito em qualquer linguagem descrita pela norma IEC 31161-3 (Texto Estruturado neste caso).

Os resultados obtidos nos testes e simulações do código de linguagem em texto estruturado realizados diretamente no equipamento (*kit* SML) e no simulador demonstram que os objetivos propostos, inicialmente no projeto, foram atingidos com sucesso. Pretendia-se cumprir com os requisitos de implementação do *kit* SML, cujos testes na interface gráfica realizada, demonstraram que é possível utilizar o simulador como teste prévio, garantindo assim o bom funcionamento do mesmo código quando implementado diretamente no autómato TSX 3721.

Em suma, podemos apresentar as principais conclusões retiradas com a realização do trabalho:

1º) Simulador de linguagem em texto estruturado para o autómato TSX Micro 3721 realizado;

- Testes realizados no *kit* e na interface gráfica, comprovam a eficiência das definições gramaticais (CFG) produzidas na construção do compilador Lex e Yacc.

2º) Estrutura gramatical hierarquizada definida e pronta a ser utilizada para geração de código noutra linguagem de programação (código C, Scilab, etc)

- O interpretador criado, responsável pela escolha da execução a realizar por cada instrução e gramática reconhecida pelo analisador sintático e semântico, contém uma estruturação que lhe permite facilmente ser programado para gerar código em qualquer outra linguagem de programação de alto nível.

5.2 Trabalho Futuro

Um dos objetivos principais no desenvolvimento do projeto de construção de um simulador para linguagem em texto estruturado é a sua utilização em meio universitário, em particular a sua utilização para o ensino académico, onde os alunos ou professores possam realizar testes e simulações prévias, para posterior implementação em autómatos e *kits* disponíveis em laboratório.

O simulador implementado apenas está planeado e testado para o autómato TSX 3721 e para o *kit* SML (*Washing Machine*). De qualquer modo, simula a linguagem em texto estruturado e está preparado para um conjunto variado de instruções típico da caracterização desta linguagem.

Podem ser definidos os principais pontos de estudo a desenvolver num trabalho futuro:

→ Trabalhar na otimização e robustez do simulador realizado, melhorando o seu comportamento, desempenho e aspeto gráfico;

→ De modo a permitir a utilização do simulador, no seu próprio computador a qualquer utilizador é interessante tornar a interface gráfica num objeto executável e independente, permitindo funcionar e correr em qualquer sistema operativo;

→ A utilização de variadas linguagens de programação é constante, nomeadamente a utilização em sistemas completamente diferentes aos usados baseados na norma IEC 31161-3. O estrutura gramatical definida pelos analisadores léxico e sintático, permite a construção de um gerador de linguagem de programação. Seria interessante gerar linguagem de alto nível (C ou linguagem com base em MATLAB ou Scilab, por exemplo), através do código desenvolvido em linguagem em texto estruturado;

→ Mesmo dentro da norma IEC 31161-3, gerar código em IL (Lista de Instruções) a partir da linguagem em texto estruturado, seria também uma mais valia para a versatilidade do produto final, permitindo usufruir das vantagens claras das linguagens textuais em programação de autómatos. As linguagens de programação com funcionamento intrínseco baseado em blocos ou diagramas (e.g FBD, LD, SFC), a serem simuladas, dependem muito do tipo de fabricante ou fornecedor a que estão sujeitas e portanto, gerar código em IL, seria mais um bom complemento à linguagem em texto estruturado;

→ Como breve descrição neste relatório foi referido a possibilidade de utilização de sinais analógicos em autómatos, em particular, a utilização para o autómato TSX3721. Um dos pontos a realizar com mais cuidado no futuro é a utilização de variáveis de memória digitais e analógicas;

→ O simulador funciona com um conjunto de variáveis de entrada e saída predefinidas. Seria interessante, consoante os nomes de endereço atribuídos pelo utilizador

durante a concessão do código de texto estruturado, alterar automaticamente os respetivos endereços na lista de entradas e saídas, permitindo assim, total subjetividade na simulação do autómato, não ficando limitado aos endereços predefinidos pelo simulador;

→ A identificação e gestão de erros num simulador de linguagem de programação é sempre um tópico de bastante importância. Poder identificar a linha ou o bloco de instruções (*statement*) onde o erro ocorreu é um começo interessante. Identificar qual é o erro é possível, mas envolve uma complexidade muito superior e tem de ser analisado com cuidado no futuro;

→ Este simulador de linguagem em texto estruturado apenas está testado especificamente para o autómato TSX3721, e como tal, é importante oferecer ao utilizador a possibilidade de escolher o tipo de autómato. Este processo envolve uma adaptação das configurações definidas no compilador de linguagem em texto estruturado.

→ Em parceria com a *Schneider Electric* seria um sonho comercializar o produto final do simulador, para utilização geral do mesmo na programação dos vários tipos de arquiteturas de autómatos.

BIBLIOGRAFIA

[1] .

[2] http://www.gordonelectricsupply.com/tsimages/SQAREDE01219_WB_21_PE_004.jpg. [Consult. Set. 2014].

[3] http://www.hindawi.com/journals/ijdsn/2013/505920.fig.003a.jpg. [Consult. Set. 2014].

[4] A. Aho, M. Lam, R. Sethi e J. Ullman. *Compilers: Principles, Techniques and Tool*. Ed. por P. E. India. 2ª Edi. (ano: 2003).

[5] R. Ahuja, P. Rastogi e A. Gupta. "Supervisory Control for Metro Station Using PLC & SCADA". Em: *International Journal of Scientific & Engineering Research*. Vol. 4. 3. (ano: 2013).

[6] "Annals of the History of Computing (IEEE) 2". Em: (ano: 1980). Cap. Programming the Mark 1: Early Programming Activity at the University of Manchester, 2: 130–167.

[7] *Command Line vs. GUI [Em linha]*. http://www.computerhope.com/issues/ch000619.html. [Consult. Ago. 2014]. (ano: 2008).

[8] K. T. I. . Control. *Introduction - Structured Text (ST) [Em linha]*. http://www.kronotech.com/ST/introduction.htm. [Consult. Set. 2014].

[9] *Detailed Description of Instructions and Functions*. PL7 Micro / Junior / Pro.

[10] P. Dighe. *Phases of Compiler Design [Em linha]*. Disponível em:http://www.durofy.com/phases-of-compiler-design/. [Consult. Ago 2014].

[11] A. Dunn. *The father of invention: Dick Morley looks back on the 40th anniversary of the PLC*. http://www.automationmag.com/features/the-father-of-invention-dick-morley-looks-back-on-the-40th-anniversary-of-the-plc.html. (ano: 2008).

[12] S. Electric. *Automation Plataform Modicon TSX Micro and PL7 Software*. Telemecanique. (ano: 2004).

[13] A. Francisco. *Autómatos Programáveis*. Ed. por ETEP. 2ª. FCT - Coimbra. Livrimpor - Lidel, (Jun. 2003). ISBN: 972-8480-07-5.

[14] P. Graham. *Hackers painters: big ideas from the computer age.* Ed. por P. E. India. O'Reilly Media, Inc., (ano: 2008).

[15] K. Gupta e K. Mourya. "Programmable Logic Controller". Em: *Initial Face Of Automation*. Vol. 2. 2. International Journal Of Advance Research In Science And Engineering. IJARSE, (ano: 2013).

[16] J. Hassan. *Structured Text Compiler Targeting XML*. (ano: 2010).

[17] *How PLCs Work*. `http://www.plcdev.com/how_plcs_work`. [Consult. julho 2014].

[18] H. Jack. *Automating manufacturing Systems with PLCs*. (ano: 2005). ISBN: 0750681329.

[19] N. Jandroep. *Don't Lose Your PLC Program [Em linha]*. Disponível em: `http://www.pidtechinsights.com/2012/11/19/dont-lose-your-plc-program/`. [Consult. Jul. 2014]. (ano: 2012).

[20] John, Karl-Heinz e M. Tiegelkamp. *IEC 61131-3: programming industrial automation systems*. (ano: 2010). ISBN: 3-540-67752-6.

[21] Knuth, D. E., Pardo e L. Trabb. "Encyclopedia of Computer Science and Technology". Em: cap. Early development of programming languages, 7: 419–493.

[22] M. Laughton e D. W. (ed). "Electrical Engineer's Reference book". Em: Newnes, (ano: 2003). Cap. 16: Programmable Controller, 7: 419–493.

[23] Lesk, M. E. e E. Schmidt. "Lex: A lexical analyzer generator". Em: ((ano: 1975)).

[24] J. R. Levine. *Flex and Bison: Text Processing Tools.* O Reilly Media, Inc., (ano: 2009). ISBN: 978-0-596-15597-1.

[25] Lucas-Nulle. *Washing Machine (24V DC) [Em linha]*. `http://www.lucas-nuelle.com/317/pid/6682/apg/3340/Washing-machine-24V-DC------.htm`. [Consultado em Agosto 2014]. (ano: 2013).

[26] M. Maggini. *Yet Another Compiler-Compiler [Em linha]*. Disponível em:`http://www.dii.unisi.it/~maggini/Teaching/TEL/slides`, Language processing technologies, [Consult. Ago. 2014].

[27] J. Martins, C. Lima, H. Martínez e A. Grau. *PLC Control and Matlab/Simulink Simulations – A Translation Approach, Matlab - Modelling, Programming and Simulations [Em linha]*. `http://www.intechopen.com/books/matlab-modelling-programming-and-simulations/plc-control-and-matlab-simulink-simulations-a-translation-approach`. ano: 2010.

[28] Microsoft. *Binding Controls to Data in Visual Studio [Em linha]*. `http://msdn.microsoft.com/en-us/library/ms171923.aspx`. [Conult. 29 Ago. 2014.

[29] R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press., (ano: 2003), p.274. ISBN: 978-0-19-927634-9.

[30] K. Morrison. *Backus Normal Form vs. Backus-Naur Form [Em linha]*. Disponível em:`http://compilers.iecc.com/comparch/article/93-07-017`. IECC explains some of the history of the two names. (ano: 1993).

[31] F. Narciso, A. Rios-Bolivar, F. Hidrobo e O. Gonzalez. "A Syntactic Specification for the Programming Languages of the IEC 61131-3 Standard". Em: *Advances in Computational Intelligence, Man-Machine Systems and Cybernetics* ().

[32] T. Nienmann. *A Compact GUIDE TO LEX YACC*. (ano: 2003).

[33] Norvell e Theodore. *A Short Introduction to Regular Expressions and Context-Free Grammars [Em linha]*. Disponível em:`http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`. P. 3-5. (ano: 2012).

[34] L. Palma. *Programação do Autómato TSX3721 - Introdução*. Universidade Nova de Lisboa - FCT - DEE. (ano: 2013).

[35] E. Parr. *Industrial Control Handbook*. Industrial Press Inc., (ano: 1999). ISBN: 0-8311-3085-7.

[36] Paxson, Vern, W. Estes e J. Millaway. "Flex: the fast lexical analyzer". Em: (ano: 2012).

[37] A. Pereira. "Conversão Automática de Código PLC para Ambientes de Simulação com Recurso a Normas Internacionais". Tese de mestrado. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, (ano: 2011).

[38] E. Perez, A. J, C. Silva e J. Quiroga. *Autómatas Programables Y Sistemas de Automatización*. Ed. por Marcombo. 2ª. Marcombo, (Set. 2009). ISBN: 978-94267-1575-3.

[39] J. R. Pinto. *Técnicas de Automação*. Ed. por ETEP. 2ª. FCT - Coimbra. Livrimpor - Lidel, (Mar. 2004). ISBN: 972-8480-07-5.

[40] J. N. Pires. *Automação Industrial*. Ed. por ETEP. 5ª. FCT - Coimbra. Livrimpor - Lidel, (Out. 2012). ISBN: 978-972,8480-31-8.

[41] *PLC Evaluation Board Simplifies Design of Industrial Process-Control systems [Em linha]*. `http://www.analog.com/library/analogdialogue/archives/43-04/process_control.html`. [Consult. jul. 2014].

[42] *PLC programming according to the IEC 61 131-3 standard in the Mosaic environment*. (ano: 2007).

[43] Z. Shao. *Compilers and Interpreters [Em linha]*. Disponível em:`http://flint.cs.yale.edu/cs421/lectureNotes/c04.pdf`. [Consult. Ago. 2014]. (ano: 1994 - 2014).

[44] S. Sisodiya. *Difference Between Compiler and Interpreter [Em linha]*. Disponível em:`http://www.engineersgarage.com/contribution/difference-between-compiler-and-interpreter`. [Consult. Ago. 2014].

[45] U. Strojny. *Formal Languages and compilers [Em linha]*. Disponível em:`http://brasil.cel.agh.edu.pl/~11sustrojny/en/compiler/`. [Consult. Ago. 2014]. (ano: 2014).

[46] *Using Bison*. [Em linha]: `http://rosemary.umw.edu/~finlayson/class/fall13/cpsc401/notes/08-bison.html`.